——Lisp処理系の作成——

Cプログラムブック

III

小西弘一
清水　剛

アスキー出版局

——*Lisp*処理系の作成——

Cプログラムブック III

小西弘一
清水　剛

アスキー出版局

はじめに

コンピュータが文字どおり数値計算専用の道具であった時代は過去のものとなりました．現在コンピュータの用途は，データベース処理，ワードプロセッシング，数式処理，機械翻訳，エキスパートシステムなど，非数値処理と呼ばれる分野に拡大しています．そしてまた，それらを統合すべき新しい学問分野として人工知能学も確立しつつあります．

その昔，数を扱う学問として出発した数学は，代数学や位相数学など抽象的構造を扱う分野へと発展した時，自然科学のための大きな武器となりました．それと同様に，非数値処理の能力，別の言葉でいえば〝知識〟を扱う能力を備えたコンピュータは，人間の思考を助けるための道具として，今後重要な存在になるはずです．それは自然科学の分野のみならず，心理学や哲学という非常に抽象的な概念を扱う分野においても大きな役割を果たすことになるでしょう．

このような状況の下で，プログラム言語にはどのような能力が求められるべきでしょうか．まず，知識という〝形の定まらないデータ〟を表現し蓄積するための方法が必要です．次に，蓄積されたその知識同士を結び付ける手段を持たなくてはなりません．このような要請に対し，最も有力かつシンプルな回答を与えるデータ構造が線形リストです．知識データを効率よく表現するために考案されたフレームシステム，あるいは意味ネットワークなどを実際のプログラムとして実現する場合にも，リスト構造は不可欠です．

C言語の高度なプログラム記述能力は，すでにCプログラムブックI，II（打越・濱野・梅原共著／アスキー出版局刊）において紹介されてきたとおりですが，実はC言語はリスト構造の扱いをもたいへん得意としています．リスト構造は，各データが他のデータへのポインタを持ち，そのリンクによってデータ相互の関係が表されることを特徴とします．そして，Cの持つ構造体とポインタは，この構造の実現に最適の組み合わせなのです．特にCにおいては，キャスト演算子を用いることにより，多種類の構造体を複合させた効率的なデータ構造が簡単に定義できるということも大きな利点です．ただし現在のところ，リスト構造や知識表現を扱うための標準ライブラリ関数は，Cにはほとんど存在していません．そのような目的に用いるプログラムは，プログラマが1から書き起こさなくてはならないのです．

そこで，本書ではC言語のリスト処理能力とその具体的な使用法を示す最良の例として，Lisp処理系を作成し，その全ソースリストを掲載しました．ここで作成するLispは，Common Lispを代表とする近代Lispに欠かせないバッククォート機能やcatch&throwによる大域脱出機構，日本語処理機能を備えた本格的なものです．本書をとおして，1人でも多くの方がリスト処理と非数値処理の手法を身につけ，プログラム作成に役立てていただければ幸いです．最後に，快く処理系を提供して頂いた，マイクロソフト株式会社，株式会社ライフボートの両社に，この場を借りて深く感謝の意を表します．

昭和61年9月　　筆者

本書を読む前に

本書ではLisp処理系の作成をおこなっていますが，Lisp言語に関する予備知識は一切必要ありません．それがどのような言語であり，どのような処理が内部でなされているのか，最初に1章～3章で解説しています．Lispを知らない読者はまずこの部分を読んでいただくとよいでしょう．

ただし，C言語に関しては，ある程度の基礎知識を備えた読者を想定して書かれています．C言語の文法など基本的な事柄については，各種入門書（弊社発行『入門C言語』など）および，本書の前巻である『CプログラムブックI』，『CプログラムブックII』を参照しておいてくださることを望みます．

本書で使用したシステムは以下のとおりです．

マシン	NEC PC-9801（RAM容量512Kバイト）
OS	MS-DOS ver.2.1
コンパイラ	Microsoft C ver.3.00［日本語版］

RAM容量に関しては，実用的にはやはり今回使用したシステム同様，512Kバイトは欲しいところですが，使用するセルやアトムなどの数を減らすことにより，RAM容量384Kバイトのシステムでも動作は可能です．

プログラムは可能な限りハードウェアに依存しないように書かれています．上記のシステム以外では，富士通FM-16βを使用した場合，およびLattice C ver.3.00をコンパイラとして使用した場合の動作を確認しています．またLattice C ver.2.1Xを使用した場合にも，数か所のソースリストの変更により動作することを確認しました(ただし機能は多少制限される)．さらに，実数演算機能や漢字処理機能など，かなりの機能を削除したサブセット版ではありますが，MSX-DOS上のMSX-C，およびCP/M上のLSI Cでも，核となる基本的な部分を動作させせることが可能です（これらのプログラム変更の詳細は「APPENDIX」参照）．

本書で作成するLisp処理系〝Will o'Lisp〟は，本書のために筆者らが言語仕様の段階から設計したものです．前述のように，本書ではまずこのLisp処理系の仕様とその動作原理の説明をおこなっていますから，Lispを知らない読者でも，各章を順を追って読んでいくことにより，Lispの動作が理解できるでしょう．Lispという言語は処理系による仕様の差異が大きいことで悪名を馳せていますが，このWill o'Lispの仕様には，本処理系のみに特有であるような機能は含めておりません．したがって，ここで得られたLisp処理系の動作に関する知識は，読者が他のLispを扱う場合にも役立つはずです．

本書の各章の構成は以下のとおりです.

1章　Lispの世界

ここではLispの言語的な特徴を述べます. 近年のAI(人工知能)研究の進展に伴って, Lispは〝人工知能のアセンブラ〟とも呼ばれるようになりましたが, なぜLispがそのような用途に向いているのかを実例とともに示します.

2章　Lispの言語仕様入門

Lispを扱った経験のない読者のためのLisp入門の章です. Lispの基本的データであるリストとアトムの話から始め, 関数定義の方法, Lispの特徴である再帰的関数呼び出しなど, Lispの基本的な使用法を説明します.

3章　Lispの動作原理

ここでは, Lisp処理系を作成するために必要となる, その内部動作を説明します. lambda binding (ラムダバインディング) による変数への値の代入など, Lispの動作は他の言語には見られない特有の方法を用いておこなわれます. これらを理解することは, 処理系作成の第1歩です.

4章　Will o'Lispの仕様

本書で作成するWill o'Lispの言語仕様をここで定義します.

5章　Lispの基本構造を作成する

4章で定義したLispの仕様はかなり大きなものです. そこで本章では, まず処理系の核となる部分を作成し, それをとおしてLisp処理系の基本的な動作を理解します.

6章　機能拡張

5章で作成した基本システムに, ガベージコレクタの追加, FILE I/O機能の整備など, 6段階の機能拡張を施していくことにより最終的なLIspシステムを完成します.

APPENDIX

5章と6章に分割されたソースリストに全体的な見通しをつけるための掲載リスト一覧, および本書で使用したMicrosoft C以外のコンパイラを用いる場合のソースリストの変更点を記載しました.

目　次

1章　Lispの世界

2章　Lispの言語仕様入門

3章 Lispの動作原理

4章 Will o' Lispの仕様

5章 Lispの基本構造を作成する

6章 機能拡張

APPENDIX

1章　Lispの世界

人工知能への関心が高まるにつれて，Lisp 言語はいま大きな注目をあびています．Lisp はなぜ，このような分野での活躍を期待されているのでしょうか．C 言語との比較において，どのような特徴を持つのでしょうか．本章では，そのような疑問に対する答を提示します．

1.1 Lispの歴史

Lispは最近になって大きな注目をあびていますが，その開発の時期は1950年代の終わり頃で，Fortranについで古い言語という肩書を持っています．動きの速いコンピュータの世界で，これだけの時間が経過するからには，大きな変化を体験せずにはいられません．Lispも初期のものとは，使用目的も実行形態も大きく変化してきています．ここではその変化の内容をおおざっぱに追ってみたいと思います．

1.1.1　Lispの曙

Lispは1950年代の終わりに，John McCarthyによって最初のアイデアが出されました．McCarthyは当初，数学の理論の研究のためにLispを考えたのですが，実際にIBM704上で動き始めたLISP1.5の処理系は，すでにある程度の実用性を持つようになっていました．LISP1.5という名称は，その時点ですでにLISP2開発の計画があったため，その途中段階として1.5のバージョン番号が与えられたものです．LISP2はしかしながらメモリ容量等，当時の計算機のハードウェアの制限に阻まれて実現することなく終わりました．

1.1.2　MacLispとInterLisp

60年代に入り2つのLispが発表され，以後これらが長期に渡ってLispのスタンダードとなりました．そのひとつはBolt-Beranek and Neumann社がPDP-1上に開発したBBN-LISPで，のちにXeroxのパロアルト研究所に移ってInterLispと呼ばれるようになったものです．もうひとつは，MITのProject MACがやはりPDP-1上に開発したMacLispで，ハッカーを育てたのはこの言語であるといわれています．これらのLispは人工知能研究に大きく貢献し，Lispは〝人工知能のアセンブラ〟という肩書を得ました．

MacLispは，まさにハッカーのためのLispで，無限多倍長精度数(Bignum)や読み込みマクロ機能，効率のよいコンパイラを備えるなど，強力な機能が売り物でした．

これに対し，InterLispの方は〝素人向きの〟Lispと呼ばれ，ユーザの使いやすさを配慮したプログラミングツールを数多く備えています．なかでも〝DWIM(Do What I Mean)〟という，ユーザのミススペルを検出して，正しいスペルを推測し，ユーザに聞き返して仕事を進めるシステムは有名です．

1.1.3 多くの処理系の出現

MacLispは，事実上のスタンダードでありながら，その仕様の文書化が遅れたことは，Lispの仕様の乱立を促す結果となりました．1970年代の後半からMacLispを拡張するプロジェクトがいろいろな場所でおこなわれるようになり，その結果，〝NIL(New Implementation of Lisp)〟や〝Frantz Lisp〟，〝Zeta Lisp〟などが作られました．これらのLispでは実用性がさらに追求され，膨大な数の関数を備えており，オブジェクト指向的要素を取り入れたものもありました．

また，この間にInterLispもウィンドウシステムなど，プログラミング環境を取り込む形でInterLisp-Dに発展しています．

この頃になると，それまでインタープリタシステムが中心だったのに対し，コンパイラが重視されるようになり，Lispの実行に適したハードウェアを持つLisp専用マシンも作成されるようになりました．実行速度が要求される分野においても，Lispは他の言語に引けをとらないものになりました．

1.1.4 Lispの標準化

一方，ユタ大学ではStandard Lispが開発され，さまざまな機種に移植されました．このLispはその名のとおり，Lispの標準化を目指しており，また数式処理システムREDUCEをその上で走らせるための土台の役割も果たしていました．REDUCEは普及に成功し，最も有名な数式処理システムとなって，発展を続けています．

1980年代になって，Lispの標準化の動きが活発になり，広範囲なLispユーザ，インプリメンタを含むグループの間で設計が進められました．何回も試案が作り直された後，1984年にCommon Lispの名でその仕様がまとめられました．また，当初の仕様には含まれませんでしたが，オブジェクト指向的な機能もInterLisp系から受け継ぎ，Common Loopsとしてまとめられています．

Common Lispはコンパイラを重視した非常に膨大な仕様のLispであるにも関わらず，すでにCommon Lispを標榜する多くのLispが作られています．

1.2 Lispの特徴

他のプログラム言語と比較した場合，Lispには多くの特有な性質があり，それがLispという言語にさまざまな肩書を与えています．ここでは，それらの意味を実際のプログラムを通して解説します．

1.2.1 Lispのプログラム

まず，Lispのプログラムの実例を見てください．これは，一種のパターンマッチングをおこなうものです．

```
(de match (x y)
   (cond ((atom x) (eq x y))
         ((atom y) nil)
         ((or (eq (car x) (car y)) (eq (car x) '?))
          (match (cdr x) (cdr y)))
         ((eq (car x) '*)
          (cond ((match (cdr x) (cdr y)) t)
                (t (match x (cdr y)))))
         (t nil)))
```

いまの段階では，まだこれがどんな動作をおこなうプログラムであるのか(それどころか，これのどこが〝プログラム〟であるのかさえ)見当もつかないかもしれませんが心配はいりません．Lispは非常にシンプルな言語です．原則さえ知れば，すぐに明確に意味がわかるようになることでしょう．

少しだけ説明すると，上記のプログラムは，*match*という関数の定義を記述したものです．先頭にある〝de〟というシンボルは，このプログラム全体が関数定義をおこなうものであることを示しています．その次の〝match〟が定義する関数の名を表し，その後ろの〝(x y)〟は仮引数を表しています．そして残りの部分が*match*の処理を記述した本体です．

この関数は，与えられた2つの引数のパターンがマッチするかどうかを判断し，その結果によって〝t(真)〟あるいは〝nil(偽)〟を返す，という動作をおこないます．

1.2.2 インタープリタ言語Lisp

それでは，このmatchという関数が，どのように実行されるのか見てみましょう．まず，前項に示したプログラムをLispに与え，*match*を定義する必要があります．プログラムが〝sample.lsp〞というファイルにセーブされているものとすれば，それを読み込んでくることによって，*match*を実行する用意は整います．

```
% (load '|sample.lsp|)    ……①
match
t
```

Lispは原則としてインタープリタ・システムです．すなわち，ある命令を入力すると，Lispはそれに対する応答をその場で返してきます．上の例では，①のような入力に対し，Lispがそれを実行し，無事に*match*が定義されたと応答しているのです．

次に，実際に*match*を使用してみます．なお，以降の実行例において，〝%〞はLispシステムが表示する入力プロンプトです．

```
% (match 'わたし 'わたし)    ……②
t

% (match 'たわし 'わたし)    ……③
nil
```

②の入力に対し，Lispは〝t〞を返してきます．tはLispでは〝真〞を表すシンボルで，*match*が〝わたし〞と〝わたし〞の2つの引数を比較した結果，両者のパターンがマッチした，ということを表示しています．同様に③では，*match*に〝たわし〞と〝わたし〞を比較させ，パターンがマッチしなかったことを表示しています．

1.2.3 リスト処理言語Lisp

ここで，*match*にちょっと変わったデータを与えてみましょう．

```
% (match '(私 は 猫) '(私 は 猫))    ……④
t

% (match '(私 は 猫) '(私 は 小猫))  ……⑤
nil
```

これらの例では，カッコにくくられた記号の集まり2つが比べられています．このようなカッコにくくられた集まりをリストと呼びます．④では2つのリストは同じ要素を持っているのでtが返され，⑤には異なる要素があるため，パターンはマッチせずnilが返されています．

Lispが主に扱うデータはこのリストです．Lispという名も，〝List Processor〟を略したものです．前に示した関数matchのプログラムも，実はひとつの大きなリストです．また①～⑤のようなインタープリタへの入力の形式もリストです．すなわち，Lispではプログラムも，プログラムが扱うデータもすべてリストなのです．このため，プログラム自身を処理するようなプログラムでも容易に記述することができます．

1.2.4　記号処理言語Lisp

関数*match*の使用例をもう少し見てください．

```
% (match '(私 ? 猫 ?) '(私 は 猫 です))
t

% (match '(私 ? 猫 ?) '(私 だって 猫 よ))
t

% (match '(私 ? 猫 ?) '(私 は 人間 です))
nil
```

*match*は〝?〟をワイルドカードとして使えるように定義されています．つまり，〝?〟が第1のリストに含まれていた場合には，第2のリストの対応する位置にどんな要素があってもパターンはマッチするのです．また，複数の要素にマッチするワイルドカード〝*〟も使えるようになっています．

```
% (match '(私 * 猫 ?) '(私 は 猫 です))
t

% (match '(私 * 猫 ?) '(私 は 軽い 機敏な 猫 なんです))
t

% (match '(私 * 猫 ?) '(私 は だから 人間 ですってば))
nil
```

リストの要素となっている〝私〟，〝猫〟，〝?〟，〝*〟などの記号はアトムと呼ばれるLispの基本的なデータです．Lispではプログラムを読み込むと，自動的にその文字列がこのような記号に

切り分けられ，保持されます．このため，Cのように文字あるいは文字列単位で処理をおこなうのにくらべ，Lispでは構文解析や数式処理などの記号処理が非常に簡単におこなえるようになっており，古来Lispはこの分野をホームグラウンドとして発展してきました．Lispが人工知能研究に深く関わってきたのも，この分野を通じてでした．

1.3 Cで書くLispの意義

本書の目的は，このLispという言語の処理系をCにより記述しようというものです．Cで書くことから，次のようなメリットを得ることができます

移植性

Cで書かれたソフトウェアは当然のことながら，アセンブラで書かれたものに比べて格段に移植が容易です．また，このことによって，ひとつのLispがさまざまなマシンで動き，そのLisp上で走るLispプログラムも修正なしでそれらのマシン上で走ることになります．Lispには方言が多いので，このことは大きな意味を持ちます．

透明性

また，アセンブラに比べてCのプログラムは読みやすさの点でも優れているので，Lispの構造を知りたいというLisp入門者には福音となります．

拡張性

構造の透明性は，ただちに拡張の容易さにつながります．システムに最初から組み込まれている関数も簡単に追加することができるので，グラフィックや通信機能などをどんどん取り込んで強力なLispに育てていくことができます．

以上はCがLispに与えるメリットでしたが，逆にLispがCを使ったプログラミング環境に与える影響を考えることもできます．その中で最も重要なのは，Lispの高い生産性でしょう．このことを考慮すると，次のようなプログラミングスタイルが浮かんできます．つまり，Cの持つ能力の中から必要なものを選んでLispに付加し，Lispですばやくプログラムを作成するという形です．つまり，これは一般の高級言語とアセンブラの関係がLispとCに置き換えられた形にほかなりません．

2章　Lispの言語仕様入門

本書はC言語によりLispインタープリタを記述することを最終的な目標とします．しかし読者の方々の中には，Cの使用経験は長くとも，Lispはよくわからないという方もまだ多いでしょう．そこでこの本章では，簡単なLispのプログラムならば理解できるよう，Lispがどのような言語であるのかの説明を，やや急ぎ足でお届けします．

2.1 リストとアトム －Lisp世界の微細構造－

ここでは，Lispを特徴づけるデータ構造，〝リスト〟と，その表記法であるS式の対応を説明します．リスト構造を知らなくても，Lispを使ってみることはできますが，Lispの動作をよりよく理解するためには，リスト構造に関する知識はなくてはならないものです．

2.1.1 線形リスト

Lispのリストを説明する前に，まず〝線形リスト〟について触れておきましょう．線形リストとは，連続した複数のデータを表現するためのデータ構造の一種です．これは，Fig. 2.1のように一定の大きさのデータにポインタを付属させ(この〝ポインタとデータの組〟をノード，またはセルと呼ぶ)，各ポインタが別のセルを指すことによって順につながれているものです．

セル
ポインタ
データ

線形リスト
データA_1 データA_2 データA_3 データA_n
データB_1 データB_2 データB_3 データB_n
(リストの終端を示す特別な値)

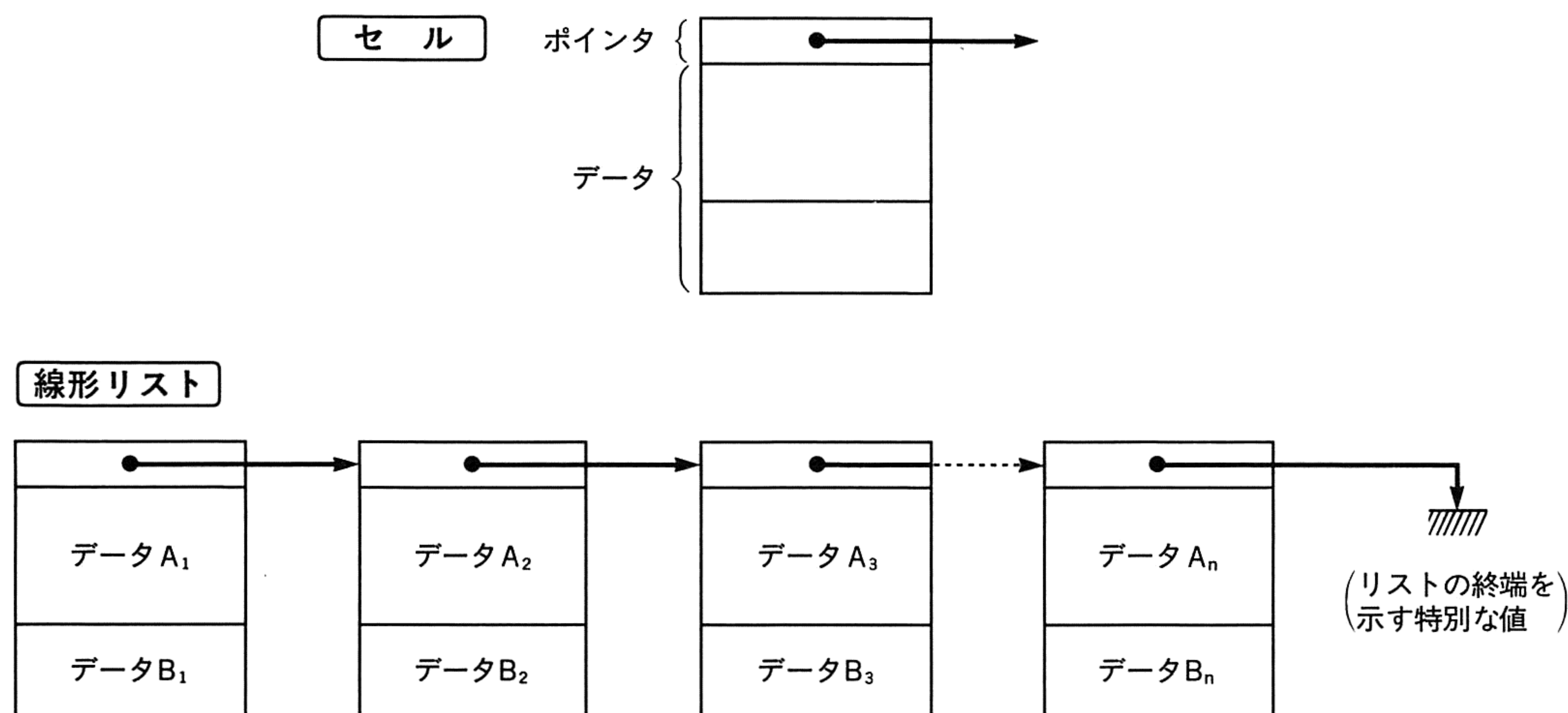



Fig. 2.1 線形リスト

線形リストの特徴は，データの追加あるいは削除をする場合に，物理的なデータの書き換えを必要としないことです．

一般に，ポインタを使用しないデータ構造では，メモリ上での物理的な位置によってデータ間の位置関係が決まります．その典型的な例が配列です．配列の場合は，あるデータに隣接して次のデータが置かれていることが本質的に重要であって，各要素はトップアドレスからの距離によって識別されます．したがって，配列のn番目の要素は，かならずその最初の要素から(1要素の

サイズ×n)バイトだけ離れたアドレスに存在していなければなりません．もし配列のある要素を更新するならば，実際にその位置にデータを転送する必要があります．さらに，すでに存在する要素の間に別のデータを挿入または削除しようとすると，それ以降のすべてのデータを前後にひとつずつ転送しなければなりません．

ところが線形リストでは，ポインタのリンクによってデータとデータの位置関係が定まります．したがって，メモリ上にいったんセルが作成されてしまえば，そのデータを線形リストに追加するのも削除するのもポインタを書き換えるだけの作業になります(Fig. 2.2)．

線形リストへのデータ追加

追加された
データ

このセルはどこに
あってもかまわない

新しいセル

線形リストからのデータの削除

削除された
データ

配列へのデータ追加

この部分を後ろへ
ずらす必要がある

←追加されたデータ

配列からのデータ削除

削除されたデータ→

この部分を前へず
らす必要がある

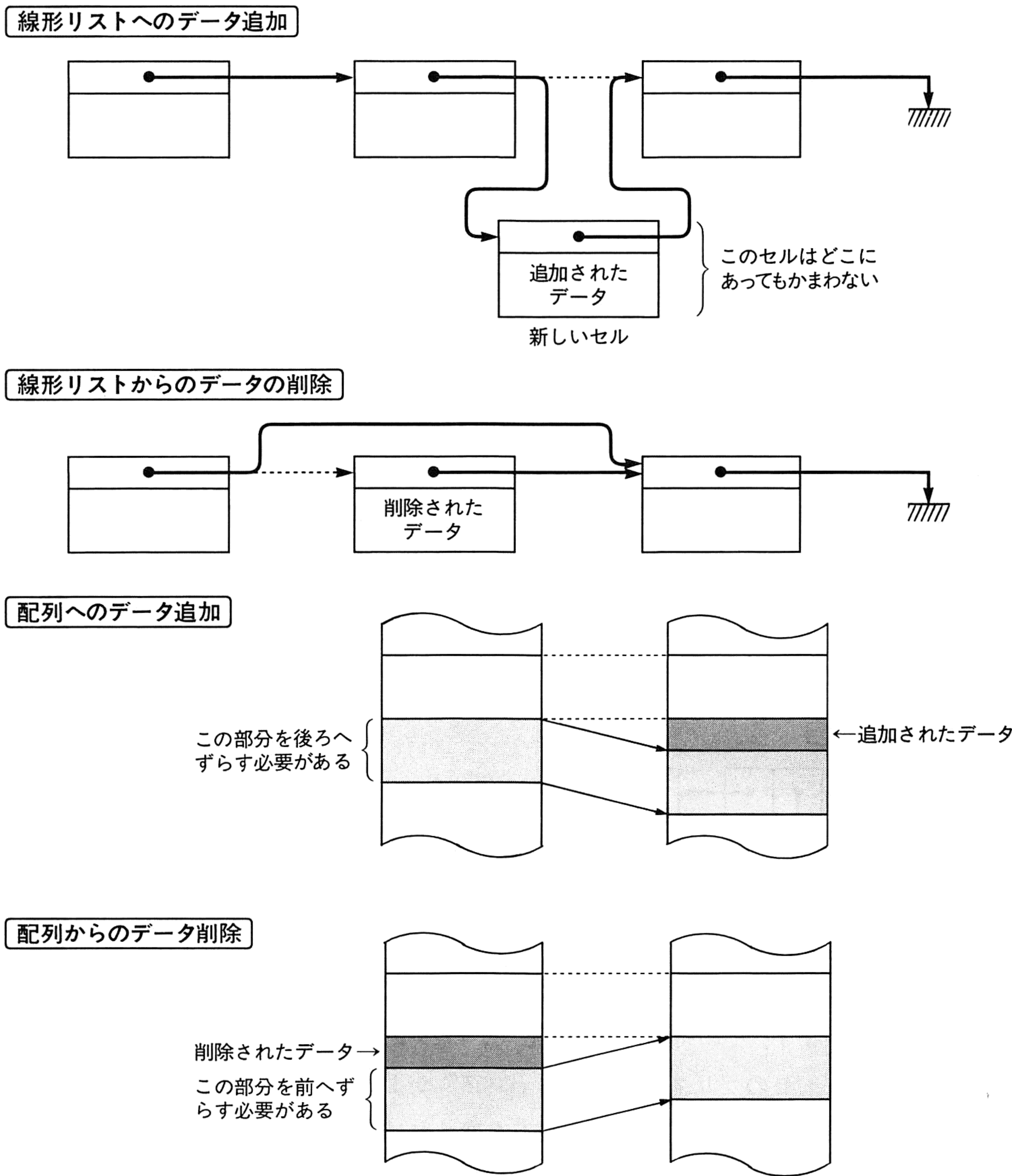



Fig. 2.2　データ構造と要素の追加・削除

一方，線形リストには，アクセススピードが遅いという欠点があります．線形リストの各要素のアドレスは配列とは違い，計算で求めることができません．外部からすぐにアクセスできるのは基本的には先頭のデータだけであり，リストの後方にあるデータは先頭のセルから順番にポインタを次々とたどっていかなければ位置がわからないのです．たとえば，1000個の要素を持つリストの最後のセルにたどりつくためには，ポインタを999回もたどらなければなりません．

したがって，線形リストのおもな使用目的は，千や万のオーダの大量のデータを一様に取り扱うことではなく，数量的には少なくとも，構造が動的に変化するデータを管理することであるといえます．

2.1.2 Lispのリスト

Lispのリストは線形リストの特殊な例で，**Fig. 2.3**のような2つのポインタからなるセルを単位としています(これを普通のセルと区別する場合はコンスセルと呼ぶ)．このセルにはデータが入る場所がなく，2つのポインタのうちのひとつがデータを指すようになっています．このポインタをcar(カー)ポインタと呼びます．もうひとつのポインタは，cdr(クダー)ポインタと呼ばれ，普通の線形リストのポインタのように他のセルを指します．リストの最後のセルのcdrポインタは，通常は〝nil〟というデータを指すことでリストの終わりを表しています．

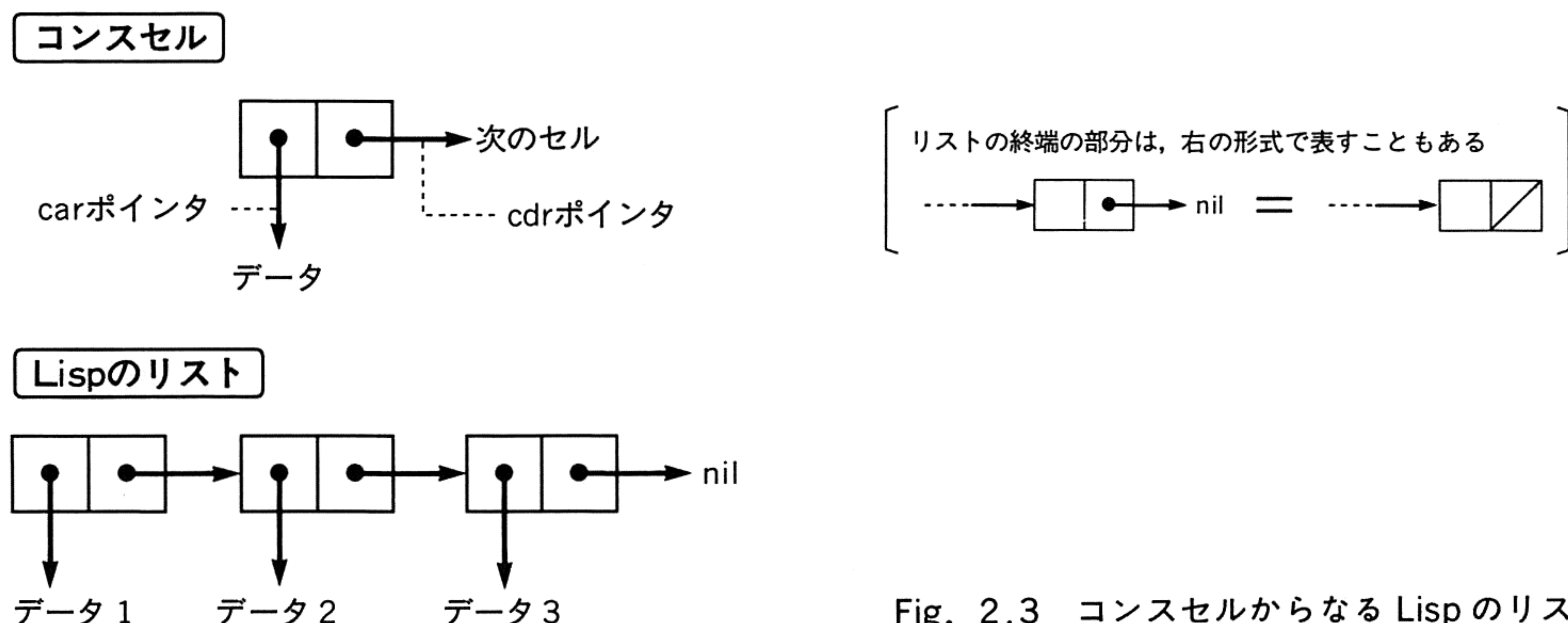



Fig. 2.3 コンスセルからなるLispのリスト

Lispのリストはデータをポインタで指すようになっているため，任意の大きさのデータを要素とすることができます．その特別の場合として，**Fig. 2.4**のようにリストを要素として含むリストも考えられます．この〝リストのネスティング〟がLispのリストの特色です．

このような階層化したリスト構造は，見方を変えれば，セルを節としてかならず二股に枝分かれする2進木構造と考えることもできます．その意味で，このような構造を持つLispのリストを2進木リストと呼びます．なお，これ以後，〝リスト〟は常に2進木リストを意味することとします．

1986 May 14

"親"のリスト

Monday 2:00

"親"の要素である"子"のリスト

AM

"子"の要素である"孫"のリスト

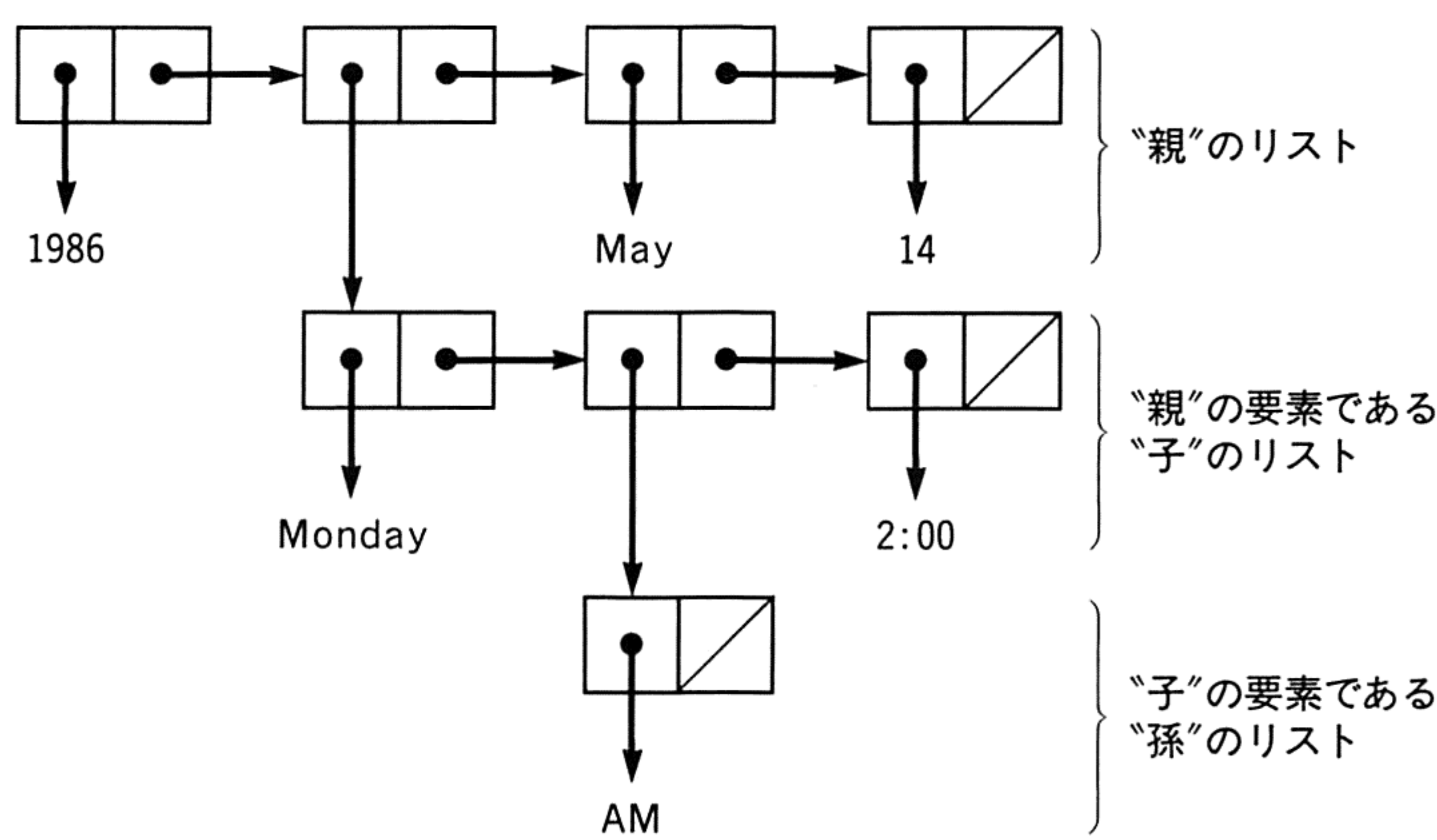



Fig. 2.4 階層化したリスト構造

●カッコによるリスト表記

さて，リストの実体は上記のようにポインタでつながれたコンスセルなのですが，実際にLispでリストを扱う際には，いちいちセルの絵を描いているわけにいきません．そこで，Lispでは，Fig. 2.5のように要素をカッコでくくることによってリストを表現します．この時もしリストの要素がリストならば，それもまたカッコでくくって表します．この表記法こそ，Lispが"カッコだらけ"になる原因です．

規　則

〈data 1〉 〈data 2〉 〈data n〉 nil

(〈data 1〉 〈data 2〉 ……… 〈data n〉)

例　1

zndoko tomoyo

(zndoko tomoyo)

例　2

mark moai zndoko tomoyo

(mark (zndoko tomoyo) moai)

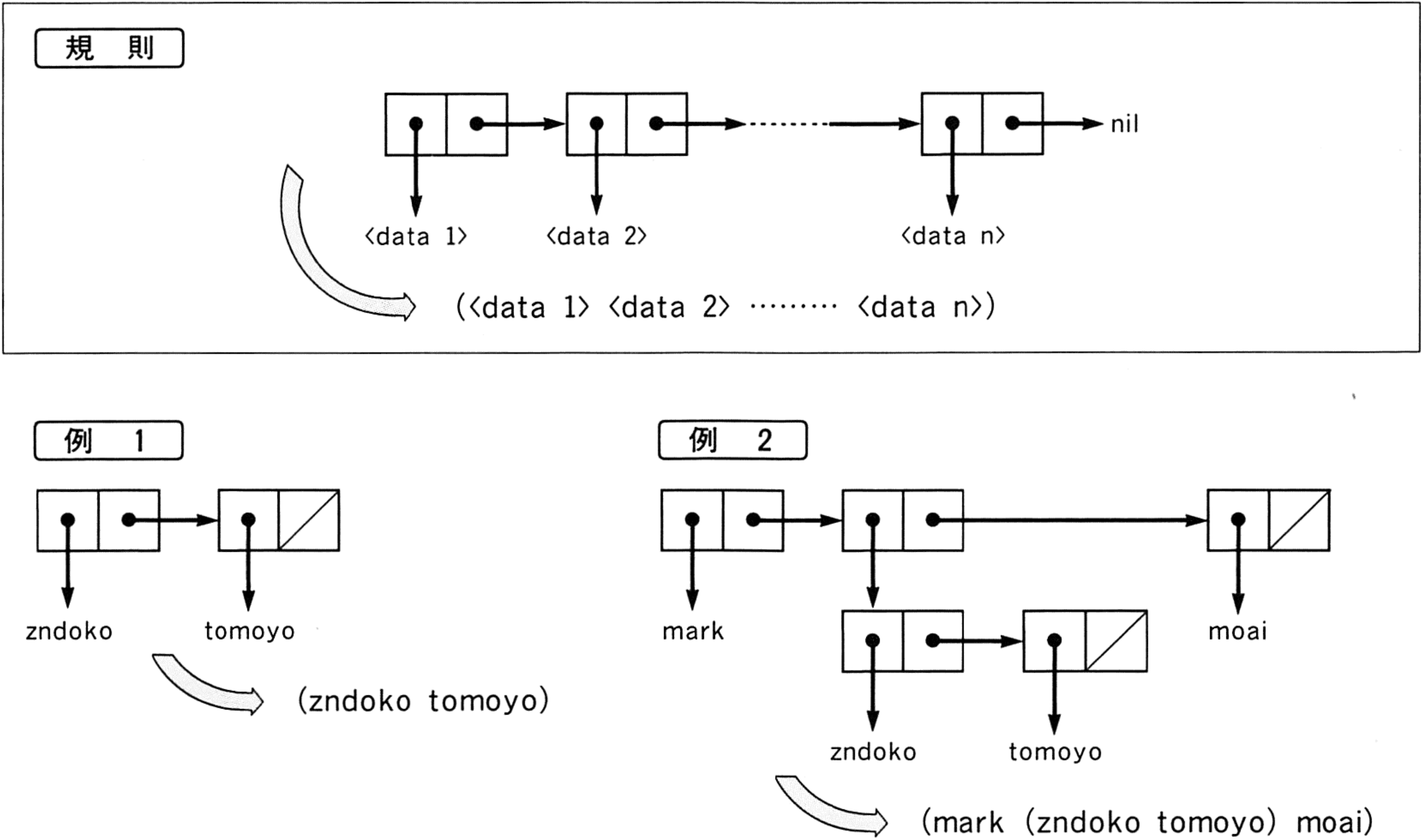



Fig. 2.5 カッコによるリストの表記法

●ドット対によるリスト表記

カッコによる表記法とは別に，Lispのリストが2進木であることを強く意識した，ドット対(dotted pair)という表現もあります．これは，Fig. 2.6のようにcarポインタが指すデータとcdrポインタが指すデータの間にドット(ピリオド)を置き，その両側をカッコで囲んで表現します．

規　則

<data 2>
<data 1>
(<data 1> . <data 2>)

例　1

breathes
dragon
(dragon . breathes)

例　2

breathes
fire
dragon
(fire . (dragon . breathes))

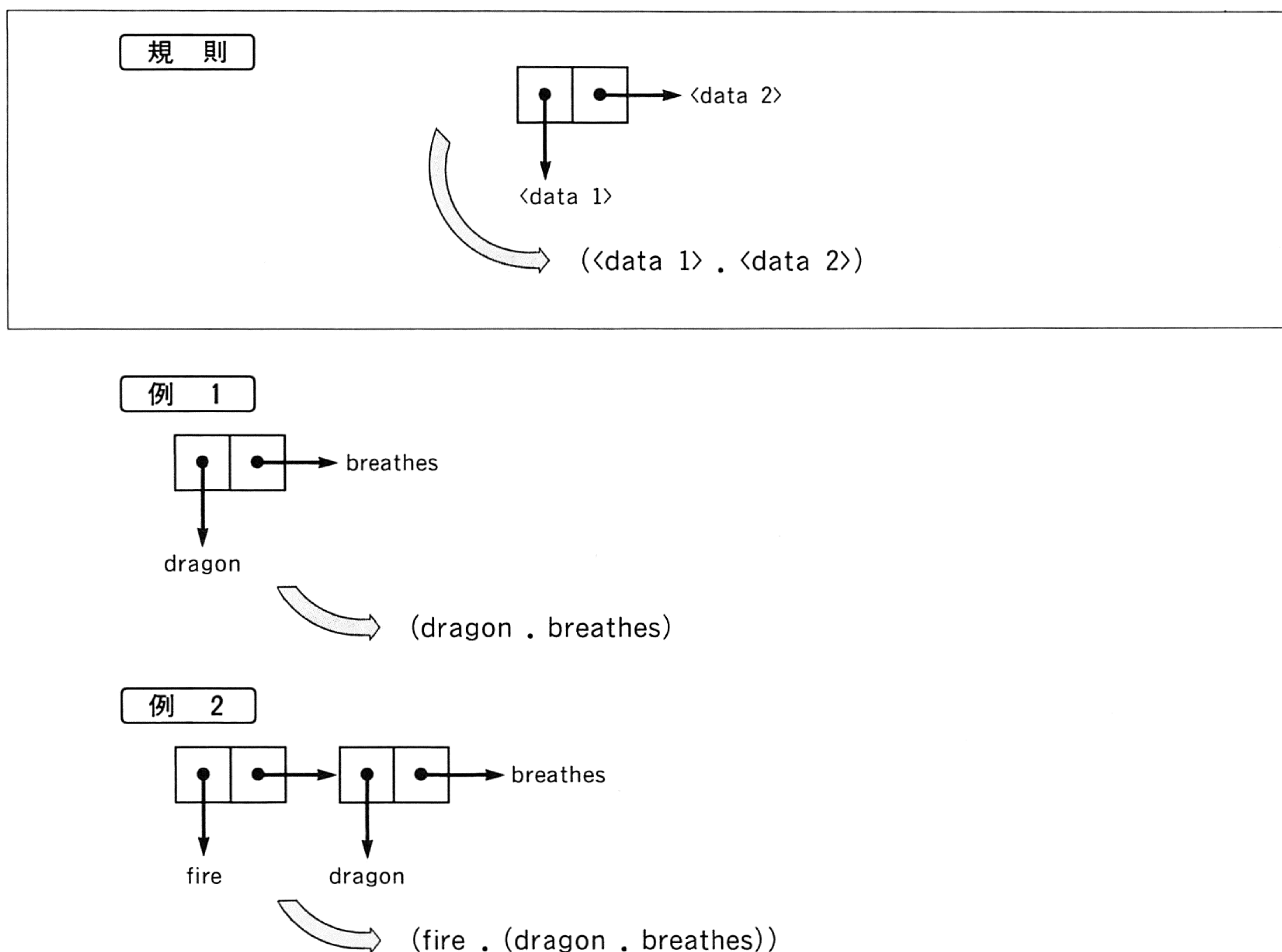



Fig. 2.6　ドット対によるリストの表記法

●混合方式のリスト表記

また，通常のリスト表現とドット対表現を混在させることもできます．この場合，どのセルに着目するかにより，Fig. 2.7のようなさまざまな表現が可能です．さらにリストの最後のセルのcdrポインタが〝リストの終端〟を示すnilを指していない場合を表現するために，Fig. 2.8のような表現も許されています．Lispシステムが出力をおこなう場合は，可能な限りドット対表現のない形，つまり通常のリスト表記か，リストの最後の部分だけをドット対の形にするFig. 2.8の方式が用いられます．

zndoko　pin　mark　nil

＝ (zndoko . (pin . (mark . nil)))
＝ (zndoko . (pin . (mark)))
＝ (zndoko . (pin mark))
＝ (zndoko pin mark)

これらすべてが上の構造を表している

Fig. 2.7　リスト表現とドット対表現の混在

規　則

<data 1>　<data 2>　<data n>　<data n+1>

(<data 1> <data 2> ……… <data n> . <data n+1>)

例

zakkai　zndoko　pin　moai

(zakkai zndoko pin . moai)

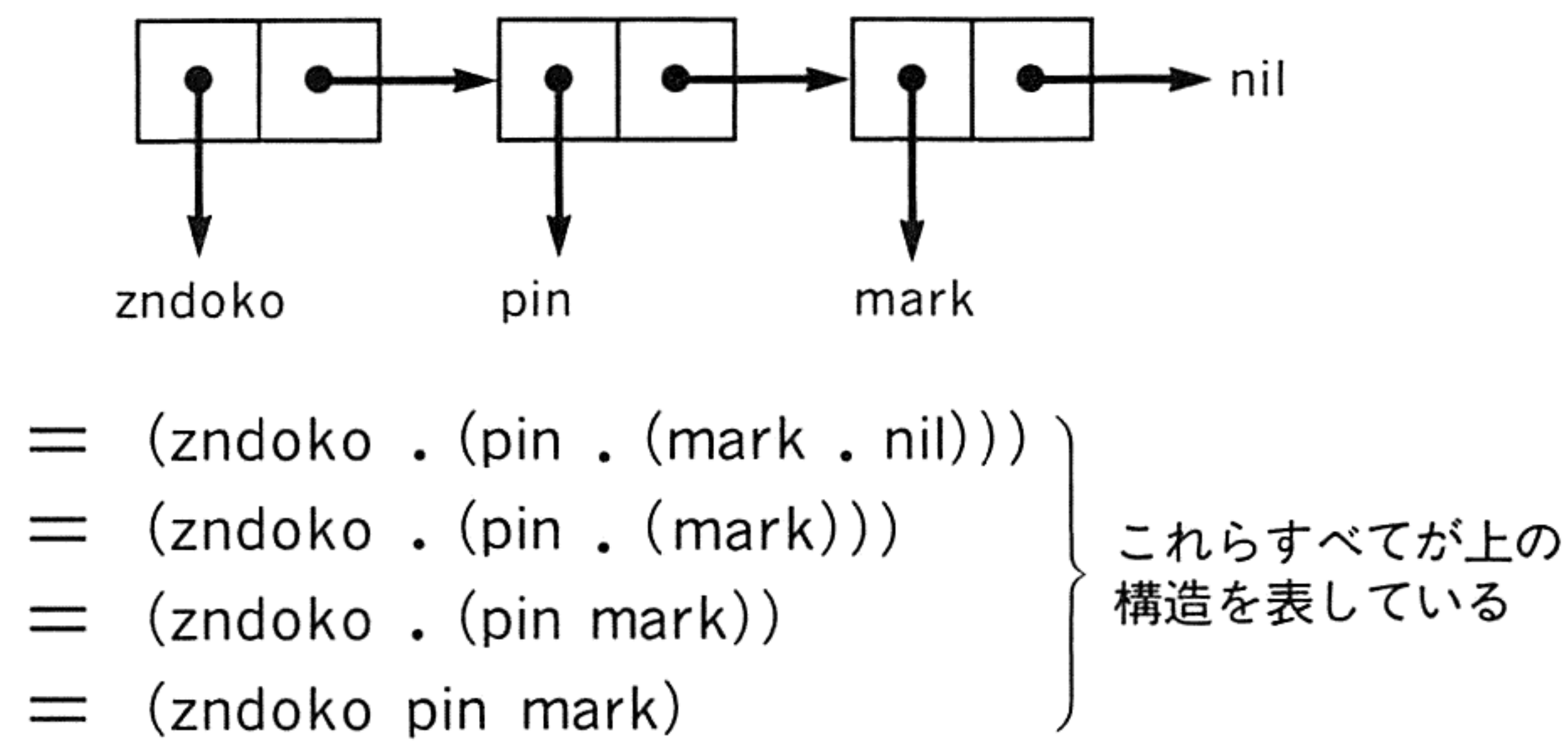
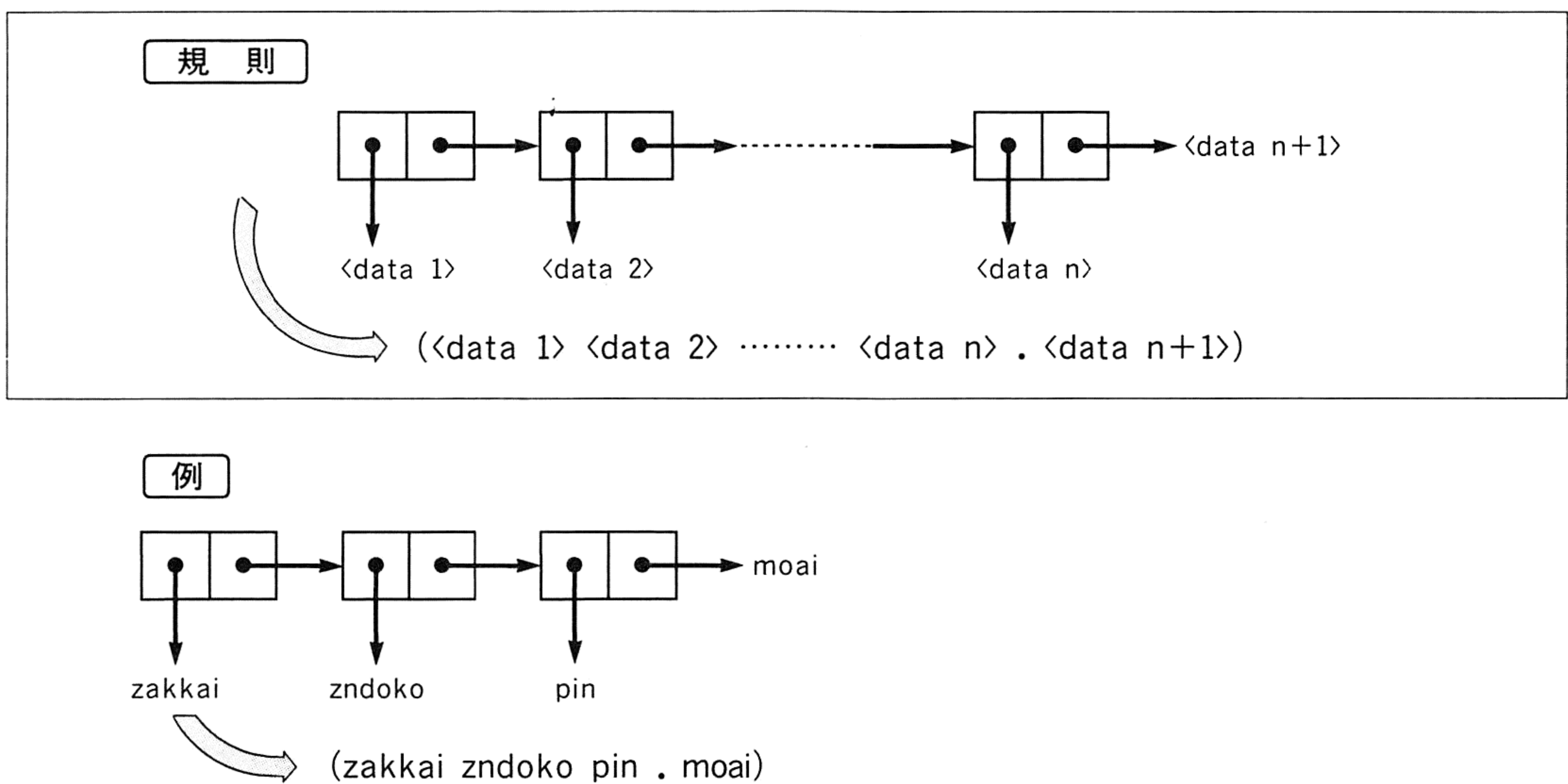



Fig. 2.8　混合表現

2.1.3 アトム

Lispの扱うデータのうち，リストでもドット対でもないものをアトムといいます．たとえば，

(plus 4 3)

というリストの中の plus，4，3や，

((pin tan) (zndoko mark) (tomoyo tam))

というリストの中の pin，tan，zndoko，mark，tomoyo，tam などはすべてアトムです．

4や3などのアトムは，数値アトムと呼びます．Lispでは数値もアトムなのです．数値アトムは整数に限らず，浮動小数点数や，LispによってはBignumと呼ばれる無限長整数のアトムも存在します．

数値アトム以外の，plusや，pin，zndokoなどのシンボルで表されているものは，数値アトムと区別する時は，シンボルアトムあるいはアイデンティファイア・アトムなどと呼ばれますが，普通は単にアトムと呼ぶことが多いようです．

●アトムの値

シンボルアトムは変数として使うことができ，任意のLispのデータを代入することができます．ただし，LispのアトムはC言語やBASICなどの変数とは違い，それ自身の中にデータを蓄える場所を備えた〝容器〞ではありません．アトムはただデータへのポインタを備えているだけなのです．そしてアトムに値を与えるというのは，Fig. 2.9のように，そのポインタの先を目的のデータに向ける操作に相当しています．

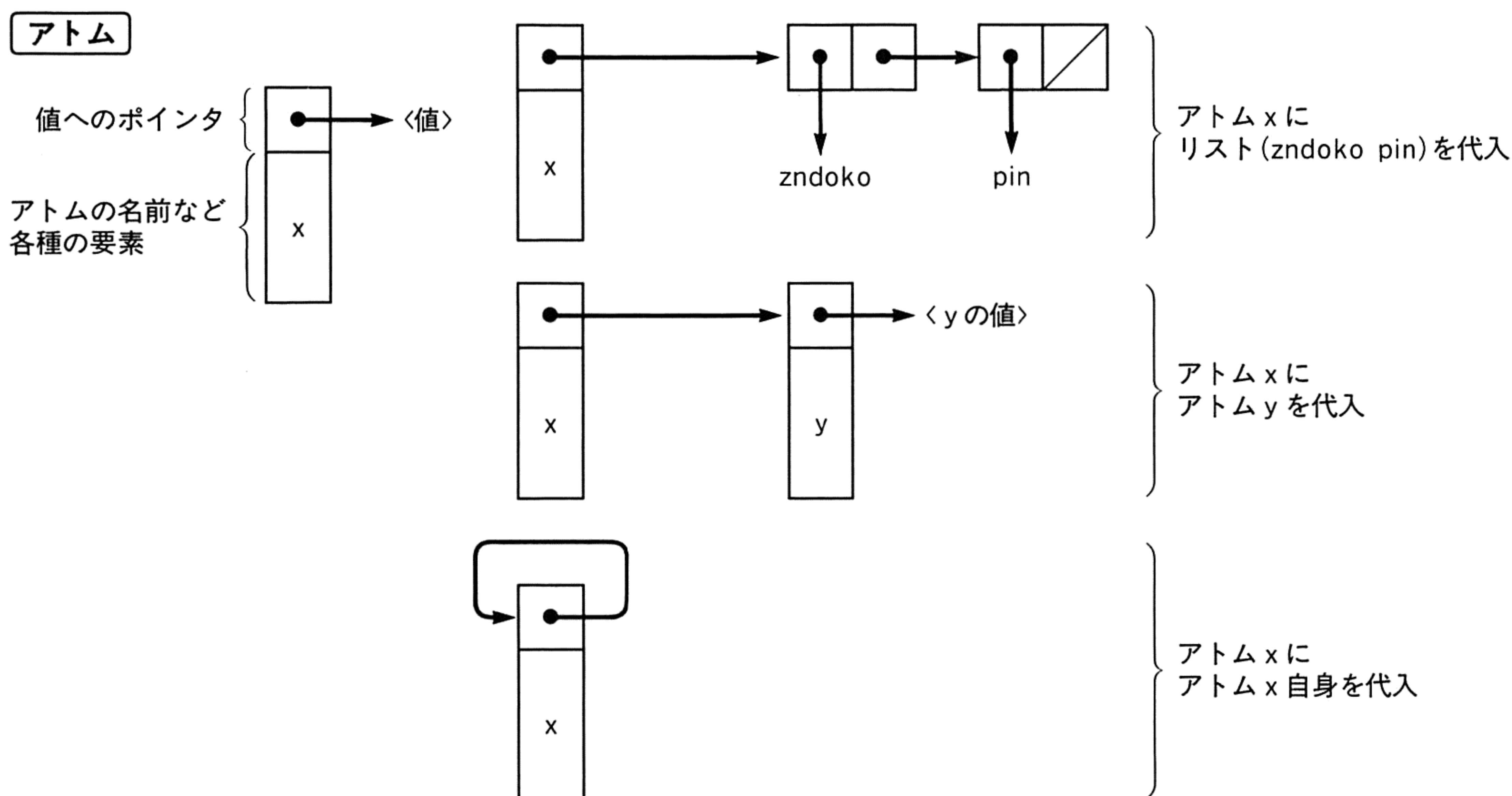



Fig. 2.9　アトムへの値の代入

少し前に，Lispのリストはcarポインタの先にデータを置く構造であるため，どのようなデータをも要素として持つことができると述べたことを思い出してください．それと同様に，アトムもポインタによって対象を指す構造であるために，どんなデータでも値とすることができます．Will o'Lispでは，作成された直後のアトムの値はそのアトム自身に設定されています．

インタープリタにアトムの名前を入力すれば，その値が出力されます．また，アトムに値を与えるためには，関数 *setq* を用い，

(setq ＜アトム＞ ＜値＞)

と入力します．この *setq* を使用した Lisp インタープリタとの対話例を次に示しましょう．なお以下の実行例において，〝%〟は Will o'Lisp のプロンプト，それに続く部分がユーザーの入力，その下の行が入力に対する応答です．

```
% apple                ……apple というアトムの当初の値は
apple                  ……apple というアトム自身である

% (setq apple 123)     ……apple に 123 を代入する
123

% apple                ……apple の値を確認すると
123                    ……123 に変更されていることがわかる
```

●アトムの属性

シンボルアトムには値とは別に〝属性〟という情報を付属させることができます．属性は普通，アトムに関する情報をそのアトム自身に持たせるために使われます．アトムに属性を与えるには，関数 *putprop* を使用し，次のようなリストを入力します．

(putprop ＜アトム＞ ＜属性の値＞ ＜属性の名前＞)

また，アトムに与えられている属性を見るためには，次に示すように関数 *get* を用います．

(get ＜アトム＞ ＜属性の名前＞)

putprop と *get* の使用例を次に示します．ここではまず，ある物体を A で表し，その材質やサイズの情報を属性および属性値として A に与えています．このように名前とその値を組として与えることにより，ひとつのアトムにいくつでも属性を持たせることが可能です．

```
% (putprop 'A '鉄 '材質)  ……物体 A の〝材質〟を〝鉄〟とする
鉄

% (putprop 'A 30 '体積)  ……物体 A の〝体積〟を〝30〟とする
30
```

さらに，〝鉄〟に関しても〝密度〟の属性を与えておきましょう．

```
% (putprop '鉄 7.8 '密度)  …… 〝鉄〟の〝密度〟を 7.8 とする
7.8
```

ここで，掛け算をおこなう関数 *times* を導入すると，物体Aの質量(＝Aの体積×Aの材質の密度)が次のように求められます．

```
% (times (get 'A '体積) (get (get 'A '材質) '密度))
234.000000
```

属性を使うと，このようにデータとデータをその間の関係を明示した形で結びつけることができます．これ以外にも，属性値の利用法はいろいろと考えられます．特に，Lispによって人工知能プログラムを作成するような場合に，知識データベースを構築するための手段として，アトムの属性はおおいに活用されています．

2.1.4 S式

リストとドット対とアトムをあわせて，〝S式(Symbolic-expression)〟といいます．S式とはすなわち，Lispで取り扱うデータの総称です．さらにLispではプログラムもすべてS式で表現されます．

ここで，S式の構文を簡単にまとめておきましょう．ただし以下の規則の中で＜数値アトム＞と＜シンボルアトム＞に関しては，自明な存在であるとして定義の記述を省略します．

＜記号の説明＞

```
[ ] : 省略可能な部分
{ } : 1回以上の繰り返し
 |  : 選択
```

＜S式の構文規則＞

```
    S式 ::= <アトム> | <リスト> | <ドット対>
  リスト ::= ( {<S式>} [ . <S式>] )
  ドット対 ::= ( <S式> . <S式> )
   アトム ::= <数値アトム> | <シンボルアトム>
```

2.2 S式の評価 —Lispの基本動作—

Lispは，S式を読んで，これをある規則にしたがって別のS式に変換して出力するシステムと見ることができます．このS式の変換は，言い換えると入力されたS式の値を求めることであり，その意味で，この変換作業を〝評価〟といいます．すなわち，LispはS式を評価するシステムなのです．ここではそのS式の変換規則を説明していくことにします．

2.2.1 アトムの評価

アトムの説明のところで，〝アトムは値を持つことができる〟と述べました．〝評価〟とは値を求めることですから，アトムを評価したものは当然そのアトムの値になります．アトムの値を見るには，ただそのアトムを入力すれば，その値が返ってくるのでした．これは，すなわちアトムを入力して，そのアトムを評価させていたわけです．

数値アトムの値はそれ自身に固定されていて変更できません．数値アトムを入力するとそのまま返ってきます．

```
% 1955   ……数値を評価すると，その数値自身となる
1955
```

また，Will o'Lispではシンボルアトムの値の初期値は，そのアトム自身になっています．したがって，アトムを入力するとそのまま返ってくるように見えます．

```
% treasure   ……シンボルアトムの初期値は，そのアトム自身
treasure
```

しかしシンボルアトムは値を変更/代入することができ，その場合には代入されている値がアトムの評価値になります．

```
% (setq treasure 12345)  ……アトム treasure に 12345 という値を代入する
12345

% treasure               ……代入された値の確認
12345
```

評価値ではなく，シンボルアトムの名前そのものが必要な場合には，アトムの直前にシングル

クォーテーションマーク“'”(クォート)をつけます．これは，その後ろに続くデータはそのままの形で扱え，と Lisp に指示するためのしるしです．

```
% 'treasure   ……クォートをつけると，シンボルアトムの値ではなく，アトム自身が得られる
treasure
```

2.2.2 関数の評価

Lisp の処理はすべて関数によっておこなわれます．ある引数に対する関数値を求めるには，関数と引数をリストにして入力します．たとえば，1+2+3 を求めるには，

```
% (plus 1 2 3)   ……関数 plus の実行
6
```

のよう入力にします．

逆にいえば，リストを入力すると，Lisp はそのリストを次のような関数と引数の組とみなしてその関数を評価するわけです．

(＜関数＞　＜引数＞　＜引数＞　……　＜引数＞)

この規則のおかげで，Lisp のプログラムはカッコだけで書くことができるのです．リストを関数として実行させず，そのままデータとして扱うには，アトムの場合と同様に直前にクォート“'”をつけます．

```
% '(plus 1 2 3)   ……クォートをつけると，リストは関数として実行されずに，リストそのものが得
(plus 1 2 3)           られる
```

関数の引数は，評価されてから関数に渡されます．

```
% (setq x 4)              ……x に 4 を代入しておく
4

% (times x (plus 3 4))    ……引数はそれぞれ評価されてから times に渡される
28
```

この例では，関数 *times* の第 1 引数 x は評価されて 4 に，第 2 引数(plus 3 4)は評価されて 7(＝3+4)になり，この 2 つが *times* に渡されて 28(＝4×7)が求まっています．引数を評価させたくなければ，やはりクォートをつけます．たとえば，(plus 3 4) というリストの car を取り出すには，

```
% (car '(plus 3 4))   ……(plus 3 4)というリストそのものを操作するためのクォートをつける
plus
```

とします.

関数の引数が評価されるということについて，もう少し説明を加えましょう．これは次の実行例に示されたように，〝引数の中に現れた関数〞の引数でも，〝そのまた中に現れた関数〞の引数でも，引数という存在でありさえすれば，どんなにネスティングレベルの深い位置にあっても評価されるということです.

```
% (difference (times 2 (plus 7 1)) 8)   ……(times 2 (plus 7 1))は differenceの引数である
8                                         から評価され，さらにその中の(plus 7 1)は times
                                          の引数であるから評価される
```

ただし，このことを〝関数の引数は最後までとことん評価される〞と拡大解釈しないように注意しなくてはなりません．たとえば次の例を見てください.

```
% (setq x '(plus 300 5))   ……前準備：アトム xの値を(plus 300 5)というリストとする
(plus 300 5)

% (times x 2)              ……まず引数である xは評価されて(plus 300 5)が得られる．次にtimes
Oops !                       はそれに 2 を掛けようとするが，〝(plus 300 5)〞は数値ではない
Error No.12……               ため，当然エラーとなる
```

ややもすると, x を評価して得られた〝(plus 300 5)〞はさらに評価されて 305 になる，と思ってしまいがちです. しかし，*times* の直接の引数となっているのはあくまでも x であり，(plus 300 5) ではありません．ですから，(plus 300 5)が評価されるべき理由はどこにもないのです.

一方＜関数＞の部分は決して評価されません．リストの先頭要素だけは他と異なる扱いを受けるのです．具体的にいうと，もし＜関数＞がアトムであれば，そのアトムに与えられている関数の定義が取り出され実行されます．また,＜関数＞部分が後述する〝lambda 式〞などの関数作用を定義する特殊なリストになっていれば, その lambda 式の定義のとおりに関数を実行します. それ以外のリストが＜関数＞の場所に現れることは許されていません.

たとえば，次の例のように＜関数＞部分に〝評価されれば正しい関数名になるS式〞を書いておいても，期待したようには実行されないので注意してください.

```
% ((car '(plus times)) 1 2 3)   ……(car '(plus times))を評価すると plusになるからこれで(plus
                                  1 2 3)の計算がおこなえるだろう，と思うのは大間違い.
                                  なぜなら, 規則によって,＜関数＞部分の S 式は評価されな
                                  いからである
```

なお，Will o'Lispを含む大部分のLisp処理系では，アトムに与えられた〝関数定義〟は，〝アトムの値〟とは別の領域に収められています．したがって，次のようなS式を評価することも可能です．

```
% (setq plus 305)     ……plusというアトムに305を代入する
305
% (plus plus plus)    ……それでも依然としてplusは加算を行う関数としての機能を失ってはいない
610
```

この例では，同じplusが3個並んだリストが入力されていますが，後ろの2つは引数として評価され305という数値が得られるだけなのに対し，先頭のplusからは，アトムplusが表す関数*plus*の定義が取り出され，それが後ろの2個の引数に適用されています．

2.2.3 特殊形式

関数のうち，特殊な機能を実現するために，通常の関数とは使い方が少し変っているものがあります．特殊な機能とは，たとえば，条件分岐や，関数の定義や，値の代入などのことです．乱暴な言い方をするならば，Cと比べた時，普通の関数の機能がCの演算子に相当するのに対し，特殊形式のそれは構文に相当します．

特殊形式は，いままでにもすでに*setq*という形で出てきています．*setq*は，アトムに値を代入する関数でした．たとえば，

```
% (setq x 'UTMC)    ……xにUTMCという値を与える
UTMC
```

とすると，アトムxにアトムUTMCが代入されるのでした．代入された値は

```
% x
UTMC    ……xの値の確認
```

として確かめることができました．

さて，ここでxの値を再び変更します．

```
% (setq x '(Urogue Totalwinner Making Club))    ……①
(Urogue Totalwinner Making Club)

% x
(Urogue Totalwinner Making Club)
```

もし，*setq* が普通の関数であれば，①における引数はすべて評価されるはずです．すると，x の値はいまは UTMC ですから，x ではなく，UTMC に後のリストが代入されることになります．実際にはそうなっていないのは，アトム x が評価されていないからです．

setq では，評価されるのは第 2 引数(代入される値)だけで，第 1 引数(代入先)は評価されません．代入先が評価されるようになっていると，プログラムの実行時に代入先を決めることもできますが，実際にはそういうことをする必要はまずありません．むしろ，代入先の変数を指定するのに，いちいち〝'〟をつけなければならないのはわずらわしいため，*setq* はこのような設定になっています．

特殊形式では引数のうちのいくつか，あるいはすべてが評価されないということが，普通の関数と異なる部分なのです．

なお，クォートを使った表記〝'<S 式>〟は，実は引数を評価させないためだけに存在する特殊形式 *quote* による〝(quote <S 式>)〟という表記を略記したものです．クォートを使用するのは，あくまでもプログラムを読みやすくするための工夫であり，Lisp の内部では *quote* 関数を使った形式で表現されています．したがって，クォートを使わずに *quote* 関数によって〝評価されない S 式〟を記述することも，もちろん可能です．

2.2.4 副作用を持つ関数

setq という関数はアトムと S 式を引数とし，S 式の値を返しますが，副作用としてアトムへの値の代入が起きます．本来の〝関数〟の定義にこだわれば，関数にとって意味を持つのは，与える引数と返される評価値の関係のみであり，代入は *setq* の副作用にすぎません．しかし，この場合副作用の方が主目的であるのは明らかです．

副作用を持つ Lisp 関数は決して例外的なものではなく，むしろ大きな Lisp システムでは多数派になっています．

2.2.5 関数の実体 —lambda 式

Lisp の関数はさまざまな形で表されます．普通に Lisp でプログラムを書く時は，シンボルアトムで表された関数を使います．

```
% (car (cdr '(wool ira tan)))
ira
```

この例では，アトム car と cdr がそれぞれ関数 *car* と *cdr* を表しています．

普通，このシンボルアトムを指して関数と呼びますが，正確に言うと，シンボルアトムは，関数の名前であって，関数そのものではありません．*car* や *cdr* の場合は，関数の実体すなわち具体的な処理内容はＣによって記述され，アトムはそのマシン語オブジェクトへのポインタを持っているわけです．Lisp で関数の実体を表現するにはどうするのかというと，lambda 式という形式が用いられます．lambda 式は次のようなリストです．

(lambda　<lambda リスト>　<本体1>　<本体2>　……　<本体n>)

lambda リストとは，lambda 式の表す関数の仮引数リストです．本体は，それを評価することによって lambda 式の関数値が得られるようなＳ式です．
たとえば，数 x に x^2 を対応させる関数は，引数の積を返す関数 *times* を使って，

(lambda (x) (times x x))

と表すことができます．これを使って３の２乗を求めるには，この lambda 式を先頭要素に持つ関数作用の形にして入力します．普通，関数の名前を置くところに，そのまま lambda 式を置くわけです．

```
% ((lambda (x) (times x x)) 3)   ……lambda 式をそのまま関数として用いる
9
```

もうひとつ，今度は x と y の２つの引数に対し，その順序をひっくり返した(y . x) というドット対を対応させる関数を考えてみます．

(lambda (x y) (cons y x))

これも次のようにして使うことができます．

```
% ((lambda (x y) (cons y x)) 'right 'left)
(left . right)
```

lambda 式には，その関数がいくつ引数をとって，それらをどのような順序で式にあてはめ，対応する関数値を求めるのかなどの情報が入っています．lambda 式は Lisp の関数の基本的な形態と言うことができます．
しかし，よく使う関数をいちいち lambda 式で書いていたのでは手間もかかりますし，ぱっと見ただけでは何をする関数かもわかりません．そこで，lambda 式で表した関数に名前をつけようということになるわけです．上の例の関数に snoc という名前をつけるのならば，関数 *de* を使って，

```
% (de snoc (x y) (cons y x))   ……①
snoc
```

と入力することになります．①を見ると，アトム de を lambda に置き変えればそのまま lambda 式になるのに気がつきます．①が入力されると，内部では上の lambda 式が実際に作られ，アトム snoc にその lambda 式へのポインタが置かれることになります．

関数に名前をつけるのは，手間を省くためばかりではありません．名前のない lambda 式だけでは再帰呼び出しをする関数を表現できないのです．たとえば，リストの長さを求める *length* という関数は，*de* および条件判断をおこなう *cond* を使えば次のように再帰的に定義できます．

```
% (de length (x)
%       (cond ((null x) 0)
%             (t (plus 1 (length (cdr x)))))) ……lengthの再帰呼び出し
length

% (length '(fig fig sam pri bis mag))
6
```

de によって，内部には，

```
(lambda (x) (cond ((null x) 0) (t (plus 1 (length (cdr x))))))
```

という lambda 式ができています．この lambda 式を length という名前を用いずに書こうとすると，length と書かれたところに代わりにこの lambda 式全体を埋め込めばよさそうなものですが，埋め込むべき lambda 式の中にも length が出てくるので，それらを順次置き換えていこうとすると，どこまでいっても終わらなくなってしまいます．

```
(lambda (x)
   (cond ((null x) 0)
         (t (plus 1 ((lambda (x)
                          (cond ((null x) 0)
                                (t (plus 1 ((lambda (x)
                                                 (cond ((null x) 0)
                                                       (t (plus 1 ((lambda (x)
                                                                        ……
```

2.3 関数の使用 ―Lispのプログラミング―

前節で述べたような〝関数〟を用いて，Lispのプログラミングがおこなわれます．ここでは，実際にプログラムを書く際に最低限必要と思われるLispの組み込み関数と，それを組み合わせて新しい関数を作り出す方法を説明します．

2.3.1 基本5関数

Lispのさまざまな関数のうち，特に基本的といわれるものがあります．*car*，*cdr*，*cons*，*eq*，*atom* の5個がそうで，この5個だけを持つLispを〝純Lisp〟ということがあります．これらの機能を順に見ていきましょう．

● car

引数のcar部を返します．したがって，引数はセルで構成されたオブジェクト(リストあるいはドット対)でなければなりません．例外として，nilのcarはnilになっています．引数がリストの場合は，与えられたリストの最初の要素を返すことになります．

```
% (car '(sug tan zakkai))          ……リストのcarは，その第1要素である
sug

% (car '(zndoko . tomoyo))         ……ドット対のcarは，その左側の要素である
zndoko

% (car 'nil)                       ……通常はアトムのcarをとるとエラーになるが，nilは例外
nil
```

● cdr

引数のcdr部を返します．したがって，引数はセルで構成されたオブジェクト(リストあるいはドット対)でなければなりません．例外として，nilのcdrはnilになっています．引数がリストの場合は，与えられたリストから最初の要素を除いたリストを返すことになります．

```
% (cdr '(ira tam tomoyo))  ……リストのcdrは，その第1要素を除いた残りの部分である
(tam tomoyo)
```

```
% (cdr '(sug . ira))          ……ドット対の cdr は，その右側の要素である
ira

% (cdr 'nil)                  ……通常はアトムの cdr をとるとエラーになるが，nil は例外
nil
```

car と *cdr* を使うと，リストのさまざまな部分を取り出すことができます．

```
% (cdr '(mark mk5))
(mk5)

% (car (cdr '(mark mk5)))                              ……リストの第2要素を取り出す
mk5

% (car (cdr (cdr (cdr '(dios dial dialma madi)))))  ……リストの第4要素を取り出す
madi
```

● cons

2つの引数をとり，car 部に第1引数を持ち，cdr 部に第2引数を持つセルを返します．第2引数がリストの時は，その先頭に第1引数をつけくわえたリストを返すことになります．

```
% (cons 'zndoko '(sin mark))  ……リストの先頭に新しい要素を加える
(zndoko sin mark)

% (cons 'Q 'M)                ……cons の第2引数がアトムの場合は，ドット対となる
(Q . M)

% (cons 'tictaco 'nil)        ……ただし，nil は例外
(tictaco)
```

cdr に置かれた nil は，リストの終わりを示すしるしなので，上の例で tictaco はドット対ではなく，リストとして出力されています．

```
% (cons (car '(no avail)) (cdr '(no avail)))  ……リスト(no avail)を car と cdr に分解し，
(no avail)                                       cons でつなぐと再び元の(no avail)になる
```

このように，あるリストの car と cdr を *cons* すると，元のリストが得られます．

● atom

引数がシンボルアトムあるいは数値アトムならばtを返し，それ以外の時はnilを返します．Lispでは，論理値の真と偽をtとnilで表します．真がtで，偽がnilです．

```
% (atom 'pin)            ……シンボルアトムに対してはtを返す
t

% (atom '(wool tan))     ……リストに対してはnilを返す
nil

% (atom 7509)            ……数値アトムに対してはtを返す
t
```

● eq

2つの引数をとり，第1引数と第2引数が同じオブジェクトであればtを，違うものならばnilを返します．ただし，次の点に注意してください．同じS式で表されているものでも，実体が異なるものを*eq*に与えると，nilが返ってくるのです．

Lispインタープリタが S式を読み込む時，シンボルアトムはoblistというシンボル登録テーブルに登録され，同じ名前のシンボルアトムには同じアドレスを対応させますが，リストや数値アトムには読み込むごとに新しいメモリを割りあてています．したがって，同じ形のリストかどうかや，同じ値の数値アトムかどうかを調べるには*eq*を使うことはできません(このような比較には，*equal*という関数を使う)．

```
% (eq 'madi 'badi)                          ……異なる名前のシンボルアトムはeqではない
nil

% (eq 'malikto 'malikto)                    ……同じ名前のシンボルアトムはeqである
t

% (eq 2541 2541)                            ……数値は，等しい値でもeqとは限らない
nil

% (eq '(zakkai wool) '(zakkai wool))        ……リストも，外見は等しくともeqとは限らない
nil
```

このように，数値アトム，リストの場合は，表記が同じものでも*eq*はtを返すとは限りません．

2.3.2 Lispの常備関数

基本5関数の他にも，ほとんどのLispに組み込まれている関数がいろいろとあります．ここではそういった，日常のプログラミングでもよく使用する，基本的な関数の説明をしていきます．

● list

リストを作る関数です．引数はいくつでもかまいません．引数すべてを要素とするリストを返します．

```
% (list 'zndoko 'wool 'tomoyo 'zakkai)
(zndoko wool tomoyo zakkai)
```

リストは *cons* さえあれば，作ることができます．たとえば，上のリストは，

```
% (cons 'zndoko (cons 'wool (cons 'tomoyo (cons 'zakkai nil))))
(zndoko wool tomoyo zakkai)
```

のように作ることもできます．しかし，この例でもわかるように，長いリストを *cons* で作るのは，記述が繁雑になります．したがって，長いリストを作るには，普通 *list* を使います．

● append

リストをつないで返す関数です．引数の数はいくつでもかまいません．

```
% (append '(tan mark sug zndoko) '(sin tomoyo))
(tan mark sug zndoko sin tomoyo)

% (append '(pin) '() '(wool) '(zakkai mk5))   ……空リストを引数に含んでもよい
(pin wool zakkai mk5)
```

● length

リストの長さを返す関数です．

```
% (length '(pin tan sug ira eof wool))
6

% (length 'nil)……通常はアトムの length をとるとエラーになるが，例外的に nil は空リストとしての
0                 意味を持つので，0 を返す
```

リストの長さ＝要素の数です．

次のように，最後のセルのcdrに入っているアトムは数に入りません．

```
% (length '(tomoyo tam mk5 . moai))  ……最後のドット対は長さには含まれない
3
```

● equal

引数はいくつでもかまいません．すべての引数が等価なＳ式である時にtを，そうでない時にはnilを返す関数です．*eq*と違って，同じアドレスにある同じオブジェクトでなくても，Ｓ式として同じ形で表されるものは等価であるとしてtを返します．したがって，*equal*を使えば，リストや数値アトムの比較もできます．

```
% (equal '(pin tan sug) '(pin tan sug))  ……リストの等価性はequalで調べる
t

% (equal '(pin zndoko sug) '(wool tomoyo mark))
nil

% (equal 1976 1976)                      ……数値の等価性もequalで調べる
t
```

● setq

いままでにも何度も出てきた，代入をおこなう関数です．引数の数は偶数であればいくつでもかまいません．奇数番目の引数に偶数番目の引数を代入します．代入先の引数は評価されません．

```
% (setq x 'zndoko y '(wool pin))  ……複数の代入をおこなえる(最後の値を返す)
(wool pin)

% x                               ……xの値の確認
zndoko

% y                               ……yの値の確認
(wool pin)
```

● print

引数をコンソールに出力する関数です．値として引数をそのまま返します．

```
% (print '(pin zndoko))
(pin zndoko)          ……printが出力したＳ式
(pin zndoko)          ……printが返した値
```

● plus

数値アトムを引数にとって，その和を求める関数です．引数はいくつあってもかまいません．

```
% (plus 1 2 3 4 5 6 7 8 9 10)  ……1+2+3+4+5+6+7+8+9+10=55
55
```

● difference

差を求める関数です．引数はいくつでもよく，最初の引数から，後の引数を次々に引いた値が返されます．

```
% (difference 32 17)   ……32−17=15
15

% (difference 12 6 8)  ……12−6−8=−2
−2
```

● times

積を求める関数です．これも引数の数は任意で，すべての引数の積を返します．

```
% (times 2 3 4)  ……2×3×4=24
24
```

● quotient

商を求める関数です．引数の数は任意で，*difference* と同様に，最初の引数を後の引数で次々に割っていった値が返されます．

```
% (quotient 32 4)    ……32÷4=8
8

% (quotient 30 2 5)  ……30÷2÷5=3
3
```

● add1

(引数+1)の数値を返します．インクリメントをおこなう関数ではないので，引数の値は元のままです．

```
% (setq x 3)  ……x←3
3
```

```
% (add1 x) ……x+1=4
4

% x          ……引数の値は変わっていない
3
```

● sub1

(引数－1)の数値を返します.

```
% (sub1 5) ……5－1=4
4
```

2.3.3 関数を定義するには

Lispでは，なんらかの仕事をするものは，すべて関数です．したがって，Lispのプログラミングとは，新しい関数を定義することをいいます．関数を定義するには次のような形式のリストを使います．

(de <関数名> <引数リスト> <本体>)

関数名は，新しく定義する関数の名前，引数リストは，仮引数として使うアトムのリスト，本体は実際の処理内容を記述したものです．

たとえば，リストの2番目の要素を取り出したいとします．すでにある関数を使っておこなう場合，そのリストから先頭要素を取り除いた残りのリストの先頭要素，すなわちそのリストのcdrのcarを取れば2番目の要素となります．

そこで，第2要素を取り出す関数*second*を定義するには，

```
% (de second (x) (car (cdr x)))
second
```

のように入力します．*second*は，

```
% (second '(one two three))       ……(one two three)の第2要素はtwoである
two

% (second '(mark tictaco tam))   ……(mark tictaco tam)の第2要素はtictacoである
tictaco
```

のように，組み込み関数の*car*や*cdr*と同様の感覚で使うことができます．

もうひとつ，複数の引数を持つ関数の例として，2次式を計算する関数を定義してみましょう．引数として，a, b, c, x をとり，ax^2+bx+c の値を求めさせることにします．これには，関数 *plus* と関数 *times* を使います．*plus* は与えられた引数全部の和を返し，*times* はやはりすべての引数の積を返します．この2つを使うと上の2次式は，

```
(plus (times a x x) (times b x) c)
```

と表せます．したがって，2次式を計算する関数 *quadratic* は，

```
% (de quadratic (a b c x)
%        (plus (times a x x) (times b x) c))
quadratic

% (quadratic 2 3 1 0.5)  ……x=0.5の時, 2x²+3x+1=3
3.000000
```

のようになります．

2.3.4 cond 式と述語関数

前述の *atom* や *eq* のように，引数がある条件を満たしているかどうかによって値が決まる関数のグループを述語関数といいます．多くは，条件が満たされた時に t を返し，そうでない時は nil を返します．

例をあげてみましょう．

plusp	引数が正の数の時 t を返す．
zerop	引数が 0 の時 t を返す．
minusp	引数が負の数の時 t を返す．
greaterp	第1引数＞第2引数の時 t を返す．
lessp	第1引数＜第2引数の時 t を返す．
atom	引数がアトムの時 t を返す．
numberp	引数が数値アトムの時 t を返す．
listp	引数がリストの時 t を返す．
null	引数が nil の時 t を返す(not として使うことができる)．

多くの述語関数の名前は p で終わっています．これは predicate(述語)の略で，いわゆるハッカー語として有名になった，〝Coffee-p？〟のように疑問文を作る p は，ここからきています．

ここで，引数の絶対値を返す関数を考えます．絶対値は，次のように場合分けによって表されます．

x≧0 の場合　‥‥　x
x<0 の場合　‥‥　−x

このような場合分けをするために，Lispには *cond* という特殊形式が用意されています．*cond* を使った次のような形式を，cond 式といいます．

```
(cond (<条件部1>  <本体1>)
      (<条件部2>  <本体2>)
          ......
      (<条件部n>  <本体n>))
```

<条件部>は，それぞれの場合を規定する条件で，<本体>は，その条件が満たされた時におこなわれる処理を記述したものです．条件部と本体の組(<条件部>　<本体>)を cond 節と呼びます．cond 式があると，Lisp は，各 cond 節の条件部を前から順に試し，最初に条件が満たされた cond 節の本体を実行して，その結果を cond 式の値とします．前にも述べましたが，Lisp では，論理値の真と偽をそれぞれ t と nil で表します．*cond* は，条件が満たされたかどうかを，条件部を評価した値が nil かどうかで判断します．この場合，t でなくても，nil でないものは真とみなされます．

では，絶対値の定義を cond 式で表してみましょう．述語関数を用いて x≧0 や x<0 を表現し，絶対値の定義をそのまま cond 式に直すと，

```
(cond ((plusp x) x)
      ((zerop x) x)
      ((minusp x) (minus x)))
```

となります．p のついていない *minus* は引数の符号をひっくり返した値を返す関数です．

しかし，絶対値の定義は次のようにも書けます．

x<0　ならば　−x
それ以外ならば　x

この〝その他の場合〟というのは，C の if 文における else や case 文の default に相当します．これを *cond* で実現するには，条件部を t とします．t は〝真〟を表すので，これに続く本体はかならず実行されます．したがって，t はその他の場合すべてをつかまえることができます．

これにより，関数 *absolute* は次のように書けます．

```
% (de absolute (x)
%      (cond ((minusp x) (minus x))
%            (t x)))
absolute

% (absolute -17)  ……|-17|=17
17

% (absolute 22)     ……|22|=22
22

% (absolute (difference 7 21))  ……|7-21|=14
14
```

2.3.5 リスト処理のプログラミング感覚

ここでは，関数 *append* と *reverse* を例にとってリスト操作の初歩を説明します．*append* も *reverse* も，たいていの Lisp では定義しなくても初めから組み込まれていますが，Lisp を使ったリスト操作のよい例なので，あえて取り上げてみました．

● append を作る

append は，引数として与えられたリストをつないだ形のリストを返す関数です．片方のリストが空リストだと，もう片方がそのまま返ってきます．

```
% (append '(a b c) '(d e f))  ……2つのリストをつなぐ
(a b c d e f)

% (appned '() '(a b c))       ……第1引数が空リストならば，第2引数が返される
(a b c)

% (append '(a b c) '())       ……第2引数が空リストならば，第1引数が返される
(a b c)
```

ところで，空リストはいったいどんな構造になっているのでしょう．Fig. 2.10 を見ればわかるように，実は，空リストは nil なのです．アトムでありながら，空リストでもあるという中途半端さが，nil の特徴です．

要素3個のリスト (zakkai zndoko pin)　zakkai　zndoko　pin　nil

要素2個のリスト (zakkai zndoko)　zakkai　zndoko　nil

要素1個のリスト (zakkai)　zakkai　nil

要素0個のリスト (空リスト) ()　nil

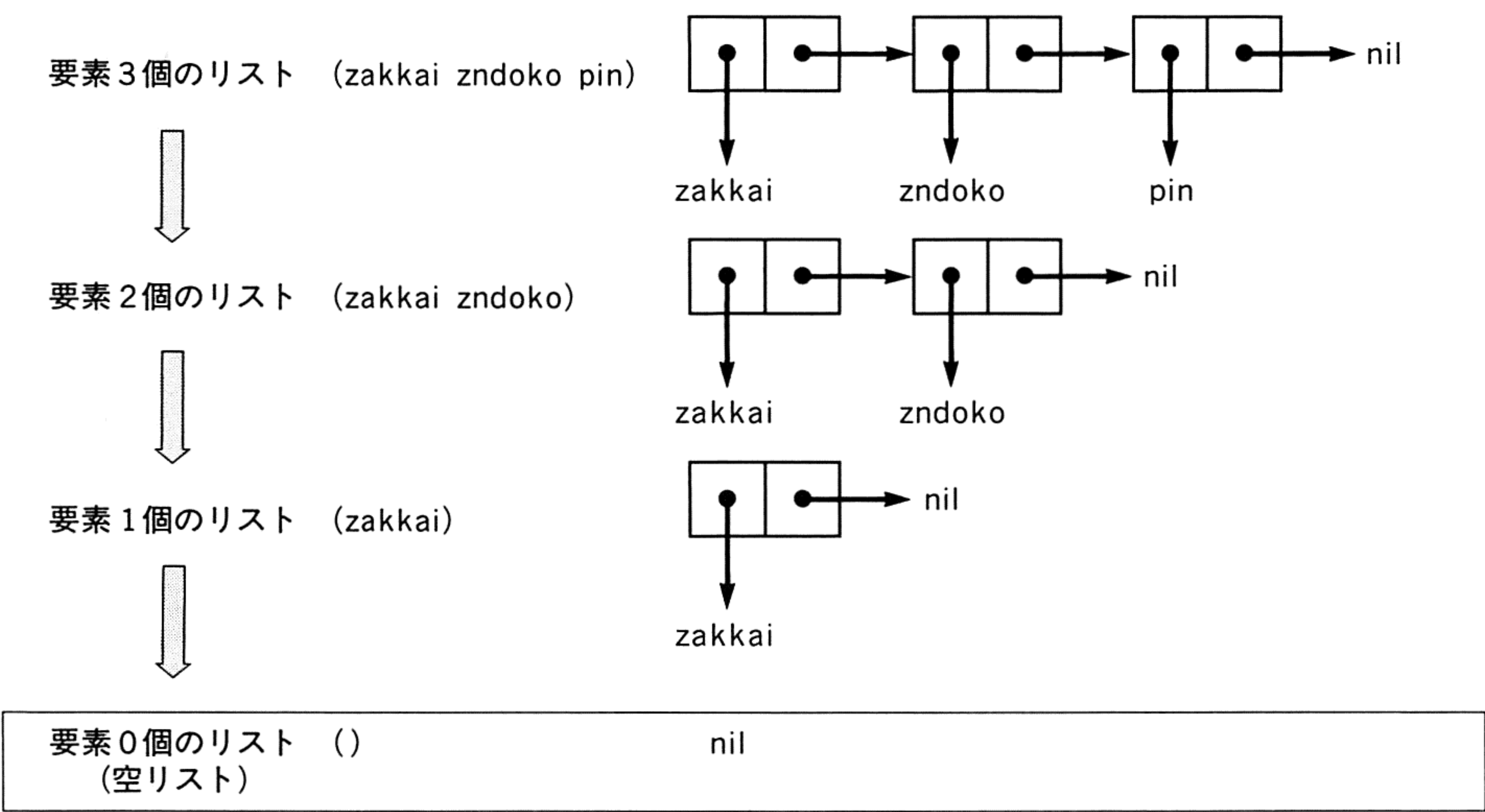



Fig. 2.10 空リスト

したがって，()の代わりにnilと書いても，同じ結果が得られます．もちろん，こんな性質を持つアトムはnilだけであり，nil以外のアトムを*append*の引数にすると，エラーになります．

```
% (append 'nil '(a b c))    ……nilは空リストと等しい
(a b c)

% (append 'a '(b c))        ……nil以外のアトムを与えるとエラーとなる
Oops !
Error: No.13 Illegal argument--List required.
At (append (quote a)(quote (b c))
```

*append*の処理は次のように分解して考えます．第1引数がnil(すなわち空リスト)ならば，第2引数をそのまま返してよいわけです．第1引数が普通のリストの時は，第1引数をcar(先頭の要素)と，cdr(carを取り去った残りのリスト)の2つに分けて考えます．求めるリストは第1引数のcdrと第2引数をappendした結果得られたリストの先頭に第1引数のcarを付け加えてやればできます(Fig. 2.11)．

*append*するために*append*しなければならないのでは，意味がないように思えますが，*append*するために*append*するために*append*するために……，と続けるにしたがって，第1引数のリストはだんだん短くなって最後にはnilになります．第1引数がnilの場合の*append*の処理はすでにわかっていますから，ここで*append*の連鎖は止まり，ここで得られる値を元にして，いままでたどってきたのとは逆順に*append*の値を求めることができるのです(Fig. 2.12)．

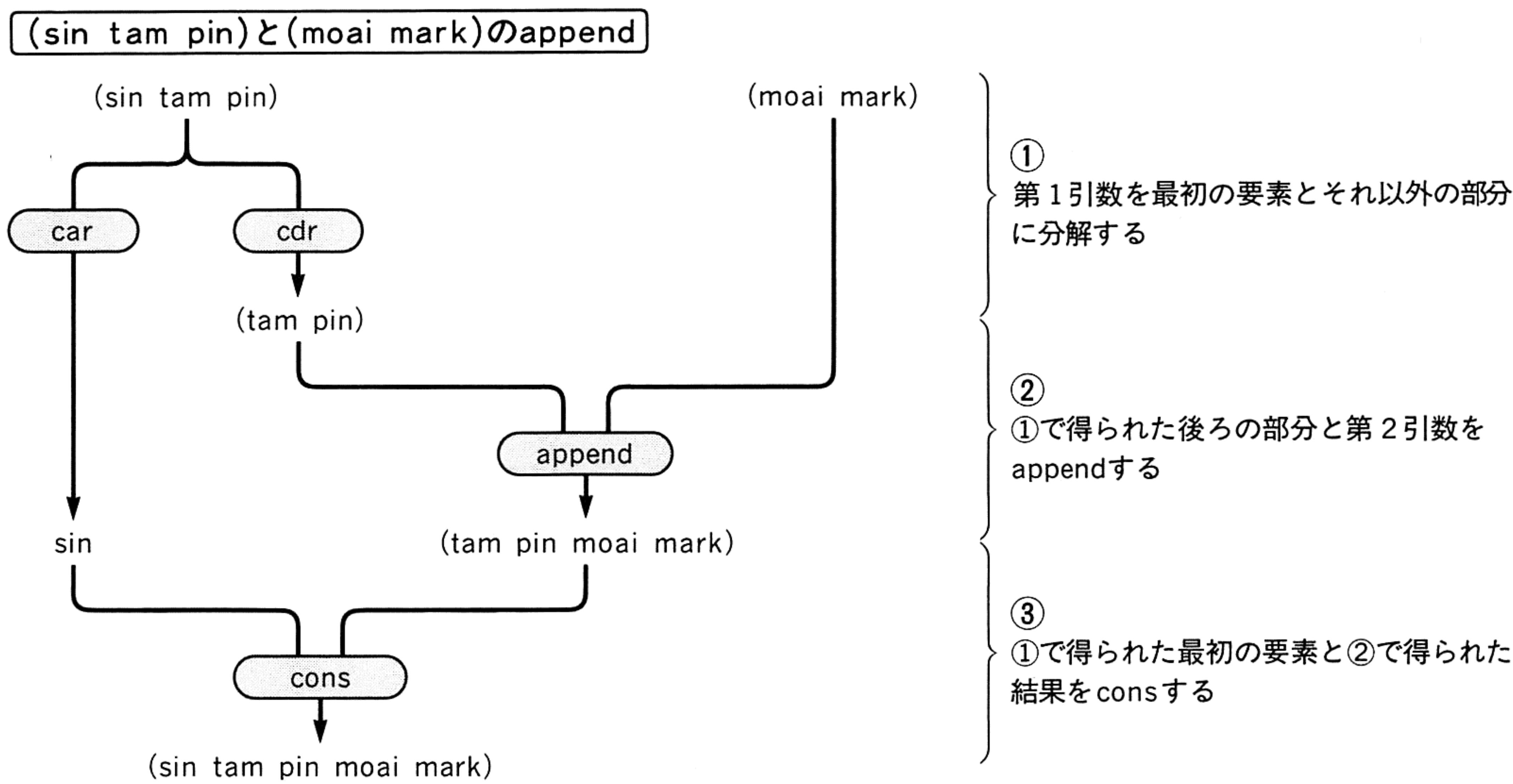



Fig. 2.11 append の処理

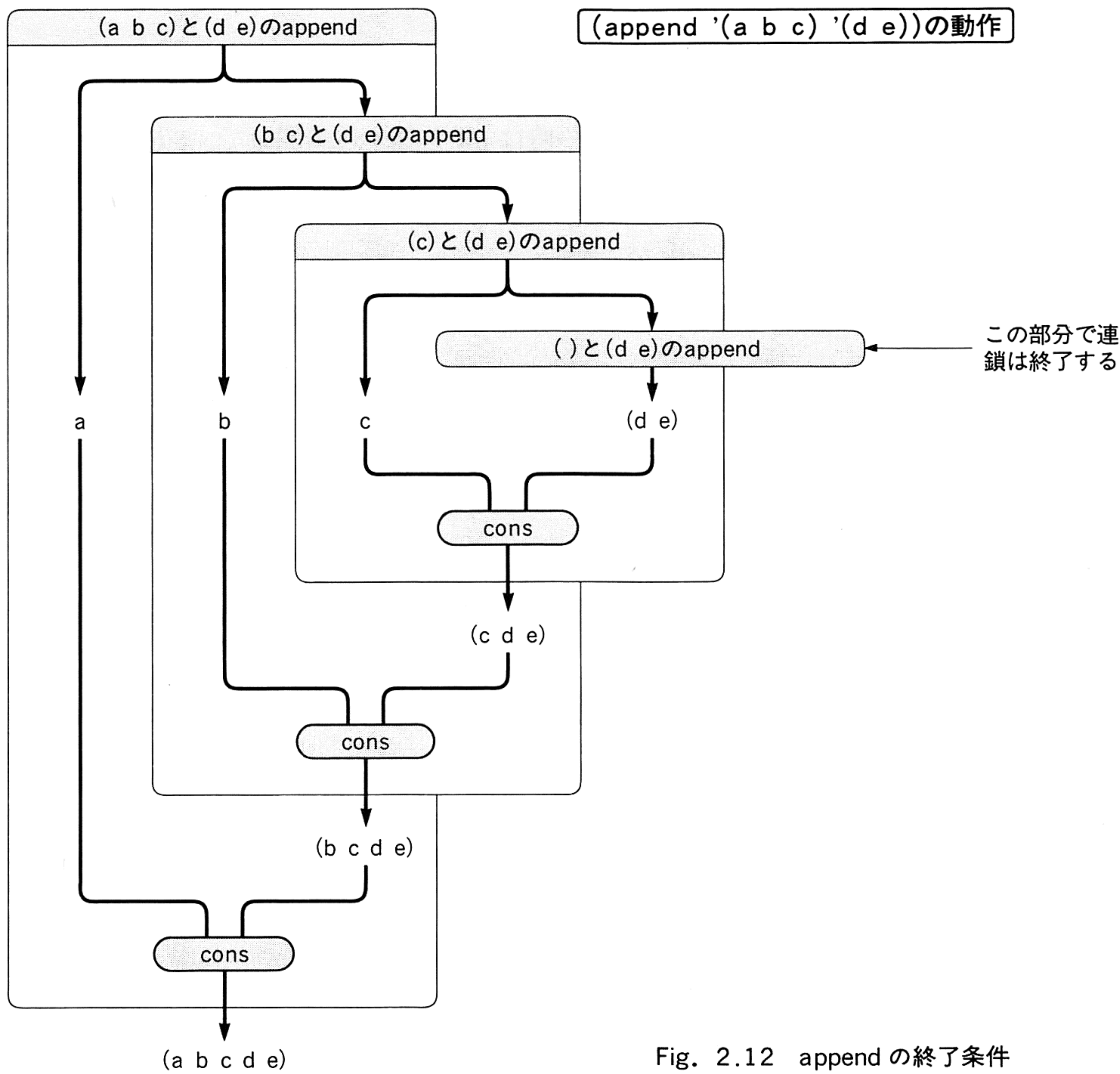



Fig. 2.12 append の終了条件

リストの先頭に要素を付け加えるには，関数 *cons* を使います．したがって，*append* の処理は，

(1)第1引数が nil ならば第2引数を返す．
(2)第1引数がリストならば第1引数の cdr と第2引数を append し，それを第1引数の car と cons する．

となります．

これで，場合分けとそれぞれの場合の処理がわかったので，cond式を書くことができます．第1引数を x，第2引数を y とすると，(1)の場合についての cond 節は，

```
((null x) y)
```

(2)の場合についての cond 節は，

```
((listp x) (cons (car x) (append (cdr x) y)))
```

となります．
ただし関数 *null* は引数が nil の時に t を返し，それ以外の時は nil を返す述語関数です．また，*listp* は，引数がリストの時 t を返し，それ以外の時は nil を返す述語関数です．以上より，cond 式全体は，

```
(cond ((null x) y)
((listp x) (cons (car x) (append (cdr x) y))))
```

となり，これにより，関数 *append* を次のように定義することができます．

```
% (de append (x y)
%       (cond ((null x) y)
%             ((listp x) (cons (car x) (append (cdr x) y)))))
append

% (append '(a b c) '(d e))  ……appendの実行例
(a b c d e)
```

なお，Will o'Lisp に最初から組み込まれている *append* は，3個以上のリストをつなぐこともできます．実際に Lisp で *append* を定義してみた方は，Lisp を立ち上げ直してから元の *append* に次のように3個以上の引数を与えて試してみてください．

```
% (append '(tam tomoyo) 'nil '(mark) '(tictaco sin))  ……組み込み関数の append は複
(tam tomoyo mark toctaco sin)                            数の引数をとれる
```

● reverse を作る

reverse は，与えられたリストの要素の順番を逆にしたリストを作って返す関数です．

```
% (reverse '(moai tam tictaco mk5))
(mk5 tictaco tam moai)
```

reverse の処理も，引数のリストを car と cdr に分けて考えます．まず，引数が空リストの時は，ない要素の順番を逆にしても元と同じですから，そのまま空リストを返せばよいわけです．引数が要素を持ったリストの場合は，求めるリストは，引数のリストの cdr だけを *reverse* したものの後ろに car をくっつけてやればできます．リストの前に何かを追加する時は *cons* すればよいのですが，リストの後ろに追加するには多少工夫がいります．ここでは，追加したいものをリストにして，*append* する方法を採ることにします．したがって，*reverse* の処理は，

(1) 引数が nil ならば nil を返す．
(2) さもなくば，引数の cdr を *reverse* したものと，引数の car をリストにしたものを，*append* したものを返す．

となります．

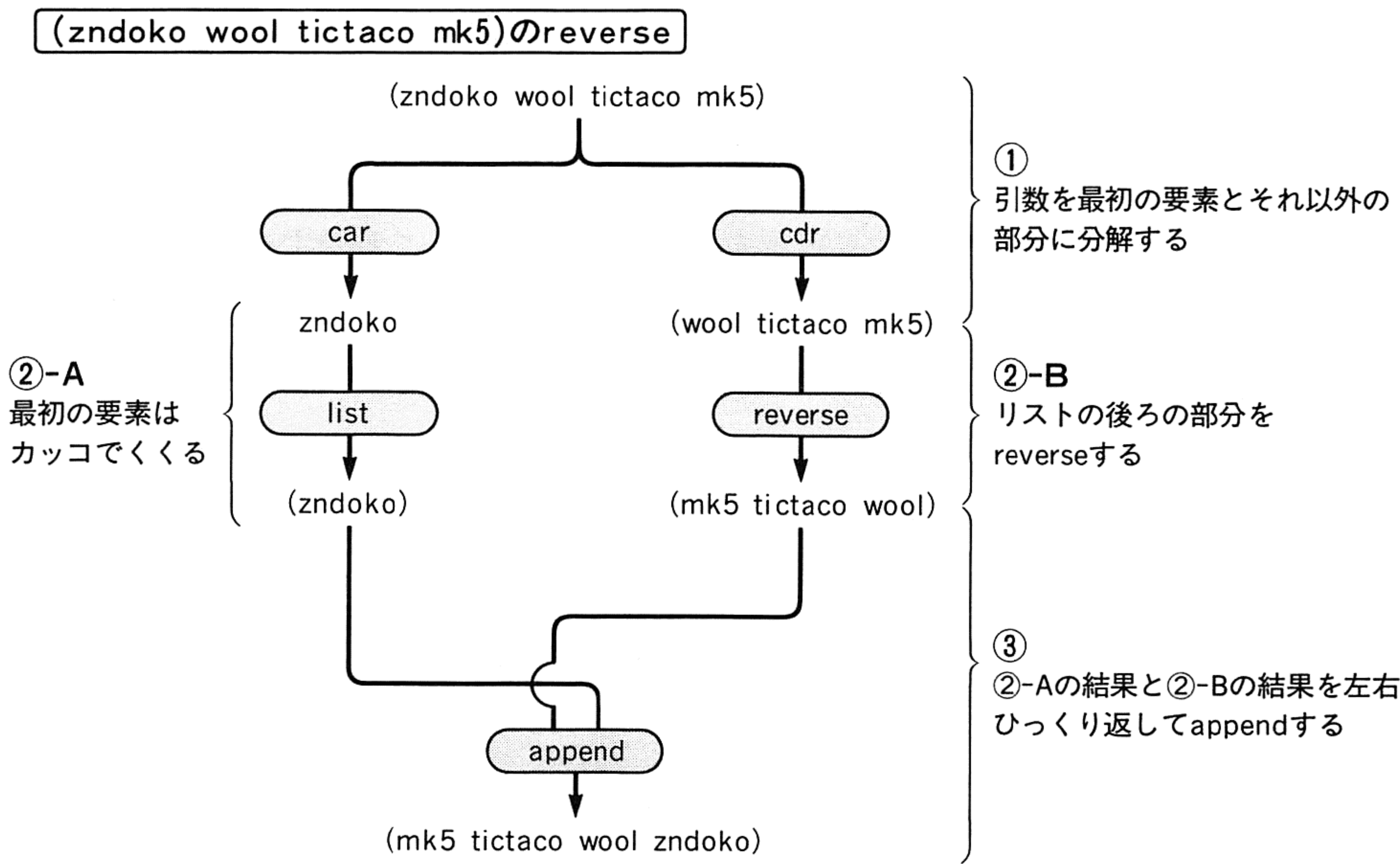



Fig. 2.13 reverse の処理

場合分けができたので，それぞれの場合を cond 節にすると，(1)は，

```
((null x) nil)
```

という形になり，(2)の cond 節は，

```
(t (append (reverse (cdr x)) (list (cdr x))))
```

となります． 以上より，*reverse* の定義の全体は，

```
% (de reverse (x)
%     (cond ((null x) nil)
%           (t (append (reverse (cdr x)) (list (car x))))))
reverse

% (reverse '(wool pin zndoko))   ……reverseの実行例
(zndoko pin wool)
```

となります．

2.4 Lisp関数の分類学

関数には，普通の関数と，特殊形式の2種類があることを前に述べました(「2.2.3 特殊形式」参照)．このほかにも，Lispの関数はさらに別の基準によって，細かく分類することができます．この違いを知っておくことは，それらの関数を使ってプログラムする時にも，またLispの処理系を作る時に，それらの関数の機能を正しく実現するためにも必要なことです．

2.4.1 関数のタイプとは

第1に，関数の処理内容が，Lispによる定義で与えられているか，Lispを記述している言語(記述言語と呼ぶ．たとえば，Will o'Lispの記述言語はC) で記述されているかによって分けることができます．システムに最初から組み込まれている関数は，記述言語で書かれています．一方 *de* を使って定義した関数の定義はS式の形で保存されています．

関数の定義は，関数 *getd* を使って取り出すことができます．たとえば，*car* の定義を取り出すと次のようになります．

```
% (getd 'car)        ……組み込み関数carの定義を取り出す
(6 . 318504968)
```

ここで返ってきているドット対は，Will o'Lisp 流の関数のタイプ番号と，C で記述された *car* を定義するルーチンのトップアドレスを表しています．これに対し，Lisp で定義した関数の定義を *getd* で取り出すと次のようになります．

```
% (de second (x) (car (cdr x)))  ……①
second

% (getd 'second)
(lambda (x) (car (cdr x)))        ……②
```

①の *de* による定義と②で返ってきたリストを比べると，先頭以外はそっくりです．すなわち，Lisp で定義された関数の定義は，②のような形で保存されているのです．

第 2 に，引数が関数に引き渡される前に評価されるものと，そのまま渡されるものに分けることができます．*cons* や *plus* などの普通の関数は，すべての引数が評価されますが，*cond* や *setq* などの特殊形式では，引数のすべて，あるいは一部は評価されません．

第 3 に，引数の個数が決まっているか，任意であるかで分類することもできます．*car* や *cons* は引数の数が 1 つあるいは 2 つと決まっていますが，*plus* は任意個の引数をとります．

以上の 3 つの基準により，関数を 8 種に分けることができます．ここでシステムに最初から組み込まれている，記述言語によって記述された関数の中から，各タイプの具体例をあげてみます．

●引数が評価され，引数の個数が一定であるもの　--- car, cons

```
% (car '(mark sin))   ……carは1つ引数をとる
mark

% (cons 'Q 'M)        ……consは2つ引数をとる
(Q . M)
```

●引数が評価され，引数の個数が不定であるもの　--- append

```
% (append '(sug ira) '(wool tan) '(pin eof))  ……3つの引数をとった例
(sug ira wool tan pin eof)

% (append '(tomoyo tam) '(mk5 moai))          ……2つの引数をとった例
(tomoyo tam mk5 moai)
```

●引数が評価されず，引数の個数が一定であるもの　　--- quote

```
% (quote tan)
tan

% (quote (zndoko wool))
(zndoko wool)
```

quote は，通常〝'〟と略記されるクォートの正体で，引数を評価せずにそのまま返す関数です．

●引数が評価されず，引数の個数が不定であるもの　　--- cond

```
% (cond ((greaterp x 0) (times x x))    ……引数(条件節)が2つ
%       (t x))
25

% (cond ((greaterp x 10) (times x x))    ……引数(条件節)は3つ
%       ((greaterp x 0) (plus x x))
%       (t x))
10
```

以上はすべてシステムに最初から組み込まれているタイプの関数ですが，Lispで定義された関数にも，やはりここで示した4つのタイプの関数があります．

2.4.2 MacLisp期の分類

Lispの黎明期，LISP1.5の時代にもすでに関数は分類されていましたが，上記のように，しつこいまでにさまざまなタイプの関数が発生したのはMacLisp期からです．MacLispでは，以上の観点から関数はTable 2.1のように分類されています．

	定義を与える言語			
	記述言語		Lisp	
引数の個数	引数の評価		引数の評価	
	する	しない	する	しない
一　定	subr	nsubr	expr	nexpr
不　定	lsubr	fsubr	lexpr	fexpr

Table. 2.1　MacLispの関数分類

MacLispではこれらの型名を指定することで，さまざまな関数を定義できるようになっています．

また，関数と似て非なるものとしてマクロがあります．マクロは，MacLisp期にさまざまな関数が現れるのと時を同じくして現れたもので，他の関数とは系列を異にする，きわめて特異な形式です(「2.5.5 マクロ」参照)．

2.4.3 Common Lisp期の分類

MacLisp以降のLispではたいてい以上のような観点による分類がおこなわれていました．ところが，Common Lisp期では，関数として生き残ったのは，〝引数の数は固定で，すべての引数が評価されるもの〟(MacLisp期のsubrとexpr)だけで，それ以外のタイプの関数は，関数ではなく特殊形式(special form)となってその数が限定され，fexprやlexprなどはなくなっています．代わりに勢力を得たのはマクロで，いくつかの特殊形式すらマクロに淘汰されてしまいました．マクロに与えた引数は評価されずにマクロに渡されます．したがってfexprのように働くマクロを作ることができます．fexprの真似ができれば，どのタイプの関数の真似もできるので，こういうことになってしまったわけです．これらの進化は効率のよいコンパイラを作りやすくしようという目的意識に導かれたものです．特殊形式の数を限定することで，徹底的な最適化をおこなえるようにし，かつマクロをコンパイル時に展開することでさらに高速化を図ることができるわけです．

2.4.4 Will o'Lispの関数の種類

Will o'Lispでは，ほぼMacLisp流の分類をしていますが，すべての型の関数は引数の個数を固定にも不定にもできるようにしてあるので，引数の個数による区別をやめています．したがって，関数は次の4種類およびマクロということになります．

subr　Cで書かれていて，引数は評価される．
expr　Lispで定義され，引数は評価される．
fsubr　Cで書かれていて，引数は評価されない．
fexpr　Lispで定義され，引数は評価されない．

Common Lispの特殊形式はほぼfexprに相当します．また，Will o'LispにはCで書かれたマクロはありません．上にそろえて書けば，次のようになります．

macro　Lispで定義され，引数は評価されない．

2.5 特殊な機能 ―効率的プログラミングツール―

ここでは，Lispプログラミングを支えている，いくつかの関数を紹介します．いままでに紹介した機能は，Lispを，遊びで使うか，研究のためにテストプログラムを作る分にはよいのですが，エディタなどのツールを作ろうとするには十分なものではありません．ここで紹介する関数は，Lispで実用的なプログラムを書くためにLispの長い歴史の中で付け加えられてきたものです．

2.5.1 prog

Lispのプログラミングスタイルは，手続き型言語に慣れた人には幾分とっつきにくい面があります．また，アルゴリズムによっては，関数型言語の再起呼び出しや引数への値渡しがなじまないこともあります．そこで，Lispで手続き型言語風にプログラムが書けるように用意されたのがprog形式です．詳しい説明はまた後で述べますが，下の2つのプログラムを見比べてみれば，説明抜きでもprog形式がどういうものかは，だいたいわかると思います．

```
circle(x, y, r)                         /* Cで記述した円を描く関数 */
int x, y, r;
{
    int a = 0;
    int b = r;
    int s = 0;
    while (a <= r/2) {
        s = s + a;
        if (s > b) {
            s = s - b;
            b = b - 1;
        }
        pointset(x + a, y + b);
        a++;
    }
}
```

```
(de circle (x y r)                        ; Lispで記述した円を描く関数
    (prog ((a 0) (b r) (s 0))
     loop (cond ((greaterp a (quotient r 2)) (return)))
          (setq s (plus s a))
          (cond ((greaterp s b)
                 (setq s (difference s b)
                       b (sub1 b))))
          (pointset (plus x a) (plus y b))
          (setq a (add1 a))
          (go loop)))
```

2.5.2 マップ関数

マップ関数は関数型言語で複数のデータに同じ処理を施すためのもので，手続き型言語のループに代わるものです．〝マップ〟とは，数学でいう写像のことです．マップ関数は，引数として，関数とリストをとります．原則的なマップ関数の作用は，引数のリストの各部分に与えられた関数を作用させ，その結果をまとめて返すという形をとります．マップ関数には，関数を作用させる対象をリストから取り出す方法や，作用させた結果をまとめる方法によってさまざまなものがあります．たとえば，そのうちのひとつ，*mapcar* は，リストの各要素に対して関数を適用し，その結果を要素とするリストを返します．

```
% (mapcar 'car '((a b) (c d) (e f)))    ……(a b), (c d), (e f)の各リストのcarを取り出す
(a c e)                                 ……取り出した要素をリストにまとめたものを返す

% (mapcar 'print '(potion scroll weapon armor))  ……potion, scroll, weapon, armorを
potion                                             printにより表示する
scroll
weapon
armor
(potion scroll weapon armor)   ……printしたものをリストにまとめたものを返す(この場合は，こ
                                  の評価値にはあまり意味はない)

% (mapcar '(lambda (x) (times x 10)) '(2 1 4 9 5))
(20 10 40 90 50)
```

最後の例では，関数を名前ではなく lambda 式の形で与えています．lambda 式は，Lisp による関数定義の実体で，この lambda 式は，引数に 10 を掛けたものを返す関数を意味しています．

2.5.3 catch & throw

たとえば，メニュー選択式のソフトウェアを考えてください．メニューのひとつを選択すると，ひとつ下のレベルのメニューが出てくるようなものです．次々にメニューを選んでメニューの階層をずっと下った所にいる時，最初のメニューが出ているところに戻りたくなったとしましょう．もし，ひとつ上のメニューに戻るという選択しかできないとすると，何度もメニュー選択を繰り返さなければ，トップメニューに戻ることができません．これを，一気にトップまで戻ってしまおうというのが catch & throw です．関数呼び出しのネストが深いところから，途中の関数の処理を省略して，一気に数層上の関数まで戻ってしまうために使われます．

```
% (de tan () (print 'tan) (mark) (print 'tanend))  ……関数 tan の定義. 中で mark を呼ん
tan                                                   でいる

% (de mark () (print 'mark) (zakkai) (print 'markend))  ……関数 mark の定義. 中で
mark                                                      zakkai を呼んでいる

% (de zakkai () (print 'zakkai) (wool) (throw 'sug 'sug) (print 'zakkaiend)) ……
zakkai        関数 zakkai の定義. wool を呼んだ後に, sug というタグへ throw をおこなっている

% (de wool () (print 'wool) (print 'woolend))  ……関数 wool の定義

% (catch 'sug (tan))  ……sug という受け手のタグを用意して, tan を呼び出す
tan
mark
zakkai
wool
woolend
sug                   ……zakkai, mark, tan を跳び越して catch へ戻った
```

上の例では，*catch* の中から関数が，*tan*，*mark*，*zakkai*，*wool* の順で呼び出された後，*wool* は正常に終了して，*zakkai* に戻っています．*zakkai* の中では，次に *throw* が起動され，ここでトップレベルへの脱出が起こります．*throw* の引数の sug は，脱出先を指定するタグで，同じタグを持つ *catch* にまで脱出がおこなわれ，*throw* の第2引数が脱出先の *catch* の値となって返されています．

2.5.4 エラートラップ

多くのインタープリタシステムでは，エラーを検出すると，エラーメッセージを出力してトップレベル(コマンド待ちをするレベル)に戻るようになっています．プログラムの開発段階では，この機能は欠かせないものですが，デバッグの終わったプログラムでは，エラーが起きるごとにいちいちプログラムを終了してトップレベルに戻っていたのでは，困ることがあります．たとえば，プログラムの中から，存在しないファイルをオープンしようとした時は，そこでプログラムを終了するのではなく，ユーザーにそのファイルが存在しないことを告げて，ファイル名の再入力を促すようになっていた方が便利です．このような希望を満たすために，トップレベルへ戻ろうとするエラーを途中で捕える関数が用意されました．

```
% (de moai () (car 'sin) (print 'moaiend))  ……エラーを含む関数 moai の定義
moai

% (moai)

Oops !
Error No.13 : Illegal Arguement--List required.  ……エラーによりトップレベルに戻ってい
at (car (quote sin))                                る

                                                                    エラーを catcherror
% (de zndoko () (catcherror (car 'sin)) (print 'zndokoend))  ……でくるんだ関数
zndoko                                                              zndoko の定義

% (zndoko)

Oops !
Error No.13 : Illegal Arguement--List required.  ……エラーが起きたが，
at (car (quote sin))

zndokoend                                          ……まだ zndoko の中にいる
zndokoend
```

関数 *moai* では，アトムに対して *car* を適用したために，エラーとなり，それに対応したエラーメッセージが出力された後，そのままトップレベルに戻っています．このことは，moaiend が出力されていないことからわかります．関数 *zndoko* では，同じエラーを *catcherror* の中で起こしているので，トップレベルには戻らず，*catcherror* まで戻って，続く *print* が実行されて zndokoend が出力されています．2つ目の zndokoend は，関数 *zndoko* の値として返されたものです．

2.5.5 マクロ

マクロは，いったん別のS式に展開されてから評価されるような形式で，Cの#define文によるマクロと同様なものです．構成されたS式をマクロ展開と呼びます．マクロを定義するには，マクロ展開を構成する関数を与えます．これを展開関数と呼びます．

例として，数値アトムからなるリストが与えられた時，その合計を算出するマクロ*sum*を作ってみます．*sum*は次のように働きます．

```
% (sum '(350 220 1200 980))  ……与えられたリストの中の数値の和を計算する
2750
```

与えられたリストの和を求めるには，各要素を取り出して順次加えていってもよいのですが，*plus*が任意個の引数をとることを利用して，

```
(plus 350 220 1200 980)
```

という形に変換できれば簡単です．そこでまずこのようなS式を作って，それを評価しようというのがマクロです．

マクロの定義は形式的には関数定義と同様です．異なるのは本体にはマクロと置き替わるべきS式を作り出す関数を置くことです．マクロにより*sum*を定義してみると次のようになります．

```
% (dm sum (x) (cons 'plus (eval x)))
sum
```

*eval*は引数を評価して返す関数で，Lispインタープリタの本体と同じものです．この定義がどう動くのかを考えてみましょう．

```
(sum '(350 220 1200 980))
```

と入力すると，仮引数xの値は'(350 220 1200 980)になります(クォートがついたままであることに注意してください)．*sum*は次のように展開されます．

```
(cons 'plus (eval '(350 220 1200 980)))
=>  (plus 350 220 1200 980)
```

このリストが*sum*のマクロ展開です．*sum*の値はこれが評価されたもの，すなわち2750になります．

2.5.6 バッククォート

マクロ展開を構成する時には，普通関数 *list* や *cons* を使います．*list* は，任意個の引数をとり，そのすべてを含むリストを返す関数で，

```
% (list 'a (car '(b c)) (plus 3 4))
(a b 7)
```

のように使います．しかし，複雑なマクロを書く時には，*list* や *cons* を何度も何度も繰り返し使わなければならず，何を書いているのかわからなくなってしまいます．また他の人が定義を見ても，さっぱりわからない，ということにもなるでしょう．

たとえば，C の if 文のように働くマクロ *if* を作ってみましょう．C の if 文は，

if（<条件部>）<実行部 1> else <実行部 2>；

という形をしており，条件部が真になった時に実行部 1 が実行され，偽になった時は実行部 2 が実行されます．これは *cond* を使うと，

(cond（<条件部> <実行部 1>）(t <実行部 2>))　……①

と書くことができます．マクロ *if* は，

(if <条件部> <実行部 1> <実行部 2>)

のように条件部と 2 つの実行部を引数にとって，上の *cond* 式のような形をしたマクロ展開を作ればよいわけです．

このマクロは次のように定義できます．

```
% (dm if (check then else)
%       (list 'cond (list check then) (list 't else)))    ……②
if
```

しかしながら，バッククォートを使えば次のようになります．

```
% (dm if (check then else)
%       ^(cond (,check ,then) (t ,else)))              ……③
if
```

cond の前にある，〝^〟が，バッククォートで，check, then，else の前にあるコンマ〝,〟は，バッククォートの中でのみ使うことのできる形式です．②と③をマクロ展開である①と比べてみれば，明らかに③の方がマクロ展開に近い形の定義になっています．ここがバッククォートのメリットです．

バッククォートの機能は，基本的にはクォートと同じで，これが前に置かれたＳ式は評価されずにそのまま扱われます．クォートと異なるのは，バッククォート形式の中のコンマに続くＳ式が評価されてからバッククォート形式全体の中に埋め込まれる点です．すなわち，バッククォートはＳ式を書くためのテンプレートを与えるものです．バッククォートを使ってマクロを書く時は，バッククォートの後に，作ろうとするマクロ展開の枠組みをそのまま書き，マクロの引数によって変化させたい部分にのみコンマをつけたＳ式を書けばよいわけです．

バッククォート形式の中で使える形式には，コンマの他〝,@〟(コンマ・アットマーク)もあります．これらを含めた詳しい機能については，「4章 Will o'Lispの仕様」および「6.6.2 バッククォート」の説明を参照してください．

3章　Lispの動作原理

2章では，Lispのプログラマにとって必要な事項を述べました．けれども，Lispを作成しようとするならば，これだけの情報で満足しているわけにはいきません．この章では，Lispのインプリメンタとして必要な事項，すなわち，Lispインタープリタがどのように S 式を処理しているのかを説明します．

3.1 Lispインタープリタの基本動作

Lisp インタープリタは，S 式を読み込んで，その S 式の種類に応じてそれを処理し，新たな S 式を作り出して出力する，という仕事を繰り返します．その意味では，一種のフィルタ型プログラムとも考えられるでしょう．ここではまず，Lisp インタープリタをその基本的な動作単位に分け，それぞれのモジュールがどのような作業をおこなっているかを説明します．

3.1.1 インタープリタの構成

Lisp インタープリタは，S 式を読み込むリーダ(reader)，読み込んだ S 式を処理し変換するエバリュエータ(evaluater)，変換された S 式を出力するプリンタ(printer)の 3 つのモジュールに大きく分けられます．そして，リーダ，エバリュエータ，プリンタの 3 者が構成しているループをトップレベルループと呼び，これが Lisp インタープリタの核になります．

ただし，実際の Lisp インタープリタは，トップレベルループを作成しただけでは完成しません．I/O やエラーへの対処など，細かい部分を記述しなくては処理系として実用にはならないからです．それらの中でも特に重要な部分が，用済みのセルやアトムを再使用できるように回収するガベージコレクタ(garbage-collector)です．

ガベージコレクタは Lisp のトップレベルループの実行に直接関与することはありませんが，それを常に監視し，セルやアトムの不足に備えていなくてはなりません．つまり Lisp インタープリタの基本的な構成は Fig. 3.1 に示すようなものとなります．

S式
リーダ
エバリュエータ
アトムおよび
セル空間
ガベージコレクタ
プリンタ
S式

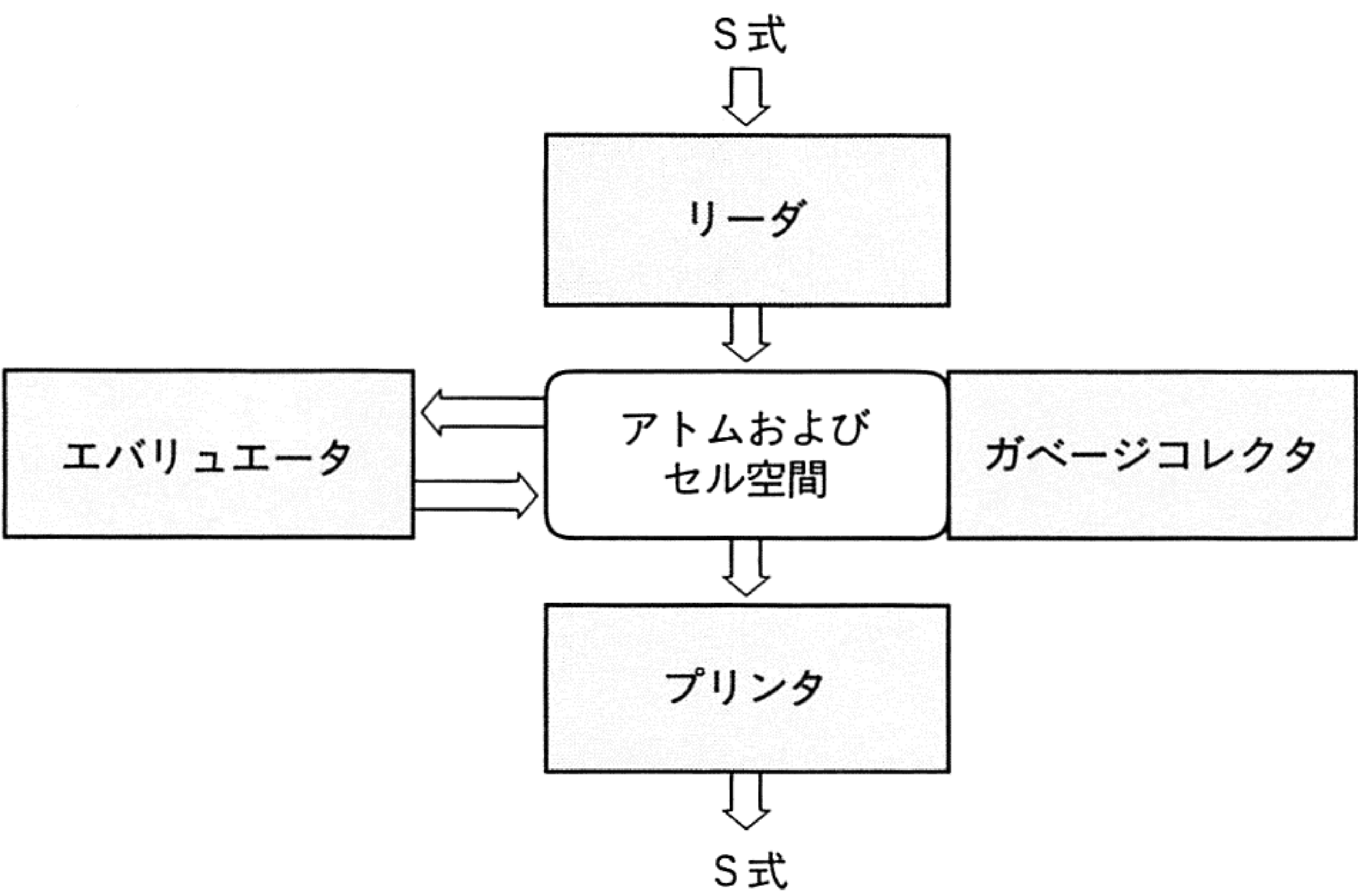



Fig. 3.1 Lisp インタプリタの基本構成

3.1.2 リーダ

リーダは，キーボードあるいはファイルからＳ式を読み込んで，それに対応するデータ構造をメモリ上に作ります．リストに対しては２進木構造を作り，アトムに対してはアトム構造体を割りあてることになります．

特に，同じ名前のアトムにはかならず同じアドレスの構造体が割りあてられるようにシンボルを管理する作業はたいへん重要です．Lisp では，このシンボル管理を oblist と呼ばれるシンボル登録テーブルを用いておこなっており，そのテーブルの管理もリーダがおこなっています．

3.1.3 エバリュエータ

エバリュエータは，Ｓ式を〝評価〟します．これについては後で詳述します．

3.1.4 プリンタ

プリンタは，メモリ上のデータ構造から，対応するＳ式を作ってコンソールあるいはファイルに出力します．プリンタはトップレベル以外に，プログラム中で使われることが特に多く，さまざまな出力スタイルを実現できるようになっていることが必要です．

3.1.5 ガベージコレクタ

Lisp はポインタの鎖によって表されるリスト構造を扱う言語であることはこれまで述べてきたとおりです．したがって新しいデータを産み出すたびに，その新しい構造のためのセルを新しく必要とします．その一方で，もはや用済みとなったセルも，たくさん作っていきます．

つまり，各々のセルは次のような３つの状態を持っています．

① 自由リストという列を作って役目が来るのを待っている状態
② リストに組み込まれて立派に役目を果たしている状態
③ 自分の所属するリストがお払い箱になり，もう使われることはないが，①の待ち行列に戻っていない〝Zombie〟の状態

ガベージコレクタは③の Zombie を捜し出して，①の自由リストに戻してやるのが仕事になります．そのタイミングに関して，使えるセルがなくなった時点でまとめて捜索するバッチ型，関数がセルを手放す時に不用になるかどうかを調べて戻していくリアルタイム型，という２種類が考

えられます．それぞれ一長一短ありますが，Will o'Lispでは前者の方法を用いることにします．いかにしてZombieを捜すかなど，その詳細については第6章で述べることにします．

3.2 S式の評価の実作業

Lispインタープリタの処理の中心は，エバリュエータの動作，すなわちS式を種類に応じて処理し，新たなS式を作り出すことです．この動作を〝S式の評価〟と呼びます．LispでS式の評価をおこなう関数にはevalという名がついており，そのことから，評価することを〝エバる〟といったりもします．そしてS式を評価してできる新たなS式を，元のS式の評価値と呼ぶことにしましょう．S式の評価値は，次の規則によって作られます．

(1) S式がシンボルアトムならば，その値を取り出す．
(2) S式が数値アトムならば，それ自身を評価値とする．
(3) S式がリストならば，〝関数作用〟とみなして評価する．すなわちリストの先頭にある要素が〝関数〟を表しているものとし，その関数定義に沿った処理が実行される．リストの2番目以降の要素は，引数として関数に渡される．
(4) '<S式>(S式の前に〝'〟がついているもの)の評価値は<S式>である．

3.2.1 アトムの値

〝アトムの評価値はアトムの値である〟というこのルールは極めて簡単ですが，ではアトムの値はどこにあるのでしょう．

2章では，アトムはそれぞれ値へのポインタを持っていて，アトムの値はその先に置かれていると述べましたが，実は，アトムの値はかならずしもアトムの中にあるとはかぎりません．アトムの値は，時に環境リストと呼ばれるリストの中に置かれていることがあります．

Lispで定義された関数を引数に適用する時には，その引数はその関数の仮引数の値になるわけです．この時，仮引数として使われているアトムは，この関数の外では別の意味を持つ変数として使われているかもしれません．その場合，仮引数となるアトムの元の値をどこかに保存しておかなければなりません．Lispによっては，この元の値を保存する方法として，仮引数への実引数の代入をアトムのポインタの書きかえではなく，環境リストを作ることでおこなっています．

環境リストは，つぎのような形をしています．

((<変数> . <値>) (<変数> . <値>) …… (<変数> . <値>))

つまりこれは変数の値のテーブルになっています.

Will o'Lisp を含めて，環境リストを使う方式の Lisp では，アトムの値をアクセスする時には

(1) 先に環境リストにそのアトムが登録されているかどうかをしらべ，あればその値にアクセスする.

(2) 環境リストの中にそのアトムが存在しなければ，アトムの中のポインタで示されている値にアクセスする.

という形になります.

変数の値と環境リストについては〝3.3 変数の有効範囲〞でさらに詳しく説明します.

3.2.2 関数作用の評価

関数作用を評価するには，

(1) 引数の評価
(2) 実引数の仮引数への代入
(3) 関数の実体の取り出し
(4) 関数の実行

の 4 つをおこなわなければなりません.

●引数の評価

引数を評価するには，普通，評価ルーチンを再帰呼び出しします. 当然ながら，特殊形式の場合には引数を評価する必要がないので，関数と特殊形式を見分けられるようにしておく必要があります.

●引数の binding

関数が記述言語で定義されている場合は，引数は直接記述言語に渡すことができますが，Lisp で定義された関数では，実引数を仮引数に代入しなければなりません.

仮引数に実引数を代入することを lambda binding(ラムダ・バインディング)といいます. Will o'Lisp では，lambda binding は，仮引数と実引数のドット対を環境リストの先頭に追加することによっておこなわれます(Fig. 3.2).

関数定義時 ←→ 関数評価時

(de f (x)………)　仮引数

(f '(a b c))　実引数

元の環境リスト

((a . 29) (y . (pin tan))………)

(x . (a b c))

新しい環境リスト

((x . (a b c)) (a . 29) (y . (pin tan))………

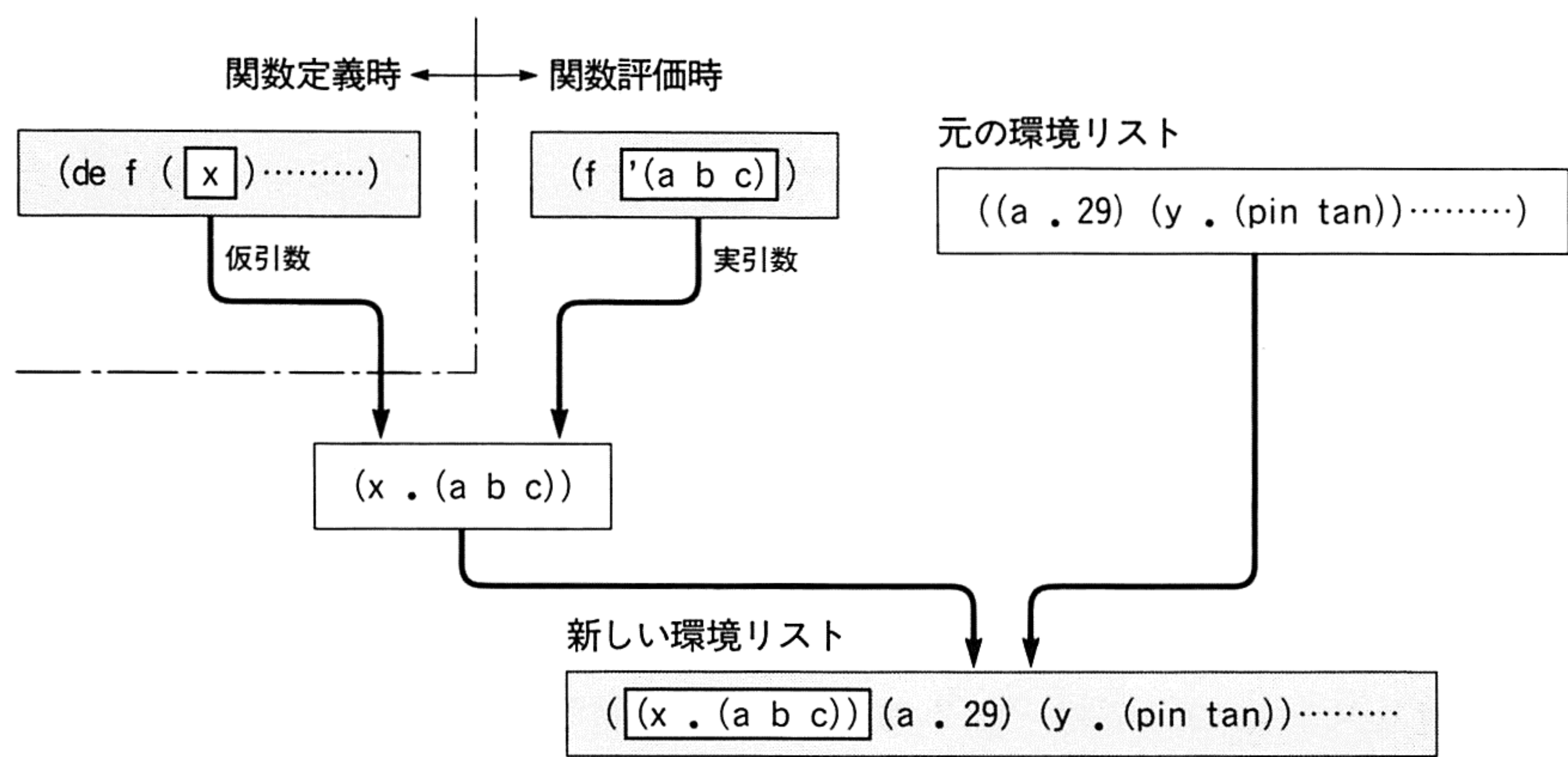



Fig. 3.2　lambda binding

アトムの値が必ずしもアトム自身の中にあるとは限らないのは，この環境リストという存在があるからです．

トップレベルでプログラマが直接setqなどでセットした値は，アトムの中に置かれます．一方lambda bindingによる仮引数の値は環境リストに置かれます．lambda bindingを起こしたlambda式の評価が終れば，そのbindingは解消されてそれによる値は消滅します．そして環境リストに値がある間は，その値がアトムの中の値に優先してアトムの評価値となります．したがって，あるlambda式の中から呼ばれた関数の中からも，その値は参照できることになります．

bindされるのはlambda式の中で仮引数として用いられるアトムだけではありません．*prog*や*let*という関数を使うと，ローカル変数を宣言することができますが，ローカル変数として宣言されたアトムは，*let*などの関数に入った時点で，与えられた初期値にbindされます．そのbindingは，やはり関数を抜ける時に解消されます．

●関数の実体はどこにあるか

関数の実体は，システムに最初から組み込まれている関数では，記述言語で書かれたルーチンであり，Lispで定義された関数ではlambda式です．関数の名前と実体を結びつける方法はLispによってさまざまですが，一般的なのは名前を表すアトムの中に，実体へのポインタを持たせておく方法です．Will o'Lispでもこの方法をとっています．

●関数の実行

関数の実体を取り出し，引数を評価したら，いよいよ関数を実行して関数値を求めなければなりません．とはいっても，これはそれほどたいした仕事ではありません．

まず，関数の実体が記述言語で与えられている場合は，これはもう何も考えずにそのルーチン

をコールするしかありません.関数値はそのルーチンが返した値をそのまま返せばよいわけです.次に関数の実体がlambda式で与えられている場合は,lambda式の本体を取り出して評価します.本体が複数ある場合は，前から順に評価していきます．そして最後の本体が返した値を関数の値として返します．本体を評価するには，S式を評価するルーチンを再帰呼び出しします．

3.3 変数の有効範囲

あるアトムが，いつ，どこで，どのような値をとるかという問題は，Lispインタープリタの動作を理解するうえで，たいへん重要です．ここでは，シンボルアトムの変数としての値を決定するルールと，それを実現するための方法を説明します．また，関数を引数にとる関数(汎関数)を使用する場合，通常のルールでは正しい変数の値が得られなくなることがあります．この問題点に対する解決法も本節で述べることにします．

3.3.1 スコープとエクステント

ある変数の空間的有効範囲をスコープといい，時間的存在範囲をエクステントといいます．これらに関する，Common Lisp仕様書に従った用語を，Cの変数を例にとって説明しましょう．

Cの関数の中で宣言された変数は，特に指定しなければ，記憶クラスautoの変数となって，その関数のローカル変数となります．その変数を参照できるのは，それが宣言された関数の構文的に(プログラムリストの字面をながめた時に)内側の部分に限られます．その関数の外からも，その関数の中で呼ばれている関数の中からもその値を見ることはできません．この場合，その変数のスコープは，その関数の内部に限定されているといい，その変数は静的スコープを持つといいます．一方，Cの関数の外で宣言されている，記憶クラスexternの変数は，グローバル変数となって，そのプログラムのあらゆるところから参照することができます．これを無限スコープといいます．

auto変数は，その関数がコールされた時にスタック上に領域が確保され，その関数からreturnする時，その領域は解放されてその値は失われます．すなわち，auto変数のエクステントは，その関数の実行中に限られています．これを動的エクステントといいます．これに対し，関数の中で宣言した変数でも，staticという記憶クラス宣言子がつけられたものは，いったんその領域が確保されるとそれは関数の実行が終わっても消えずに残り，次回その関数がコールされた時，前回の実行結果による値がそのまま参照できます．これを無限エクステントといいます．

もうひとつ，無限スコープと動的エクステントを持つ場合，これを動的スコープといいます．ある関数のローカル変数が動的スコープである場合，その変数はその関数の構文的内部だけでなく，その関数から呼ばれている関数の中からも値を参照できますが，関数の外からはその変数の値は見えません(Table 3.1)．

	空間的	時間的
関数の中だけ	静的スコープ	動的エクステント
制限なし	無限スコープ	無限エクステント

動的スコープ：無限スコープかつ動的エクステント

Table. 3.1　スコープとエクステント

以上の用語を使うと，Will o'Lisp の lambda 式の仮引数や *let* 等の関数で宣言されるローカル変数は，無限スコープと動的エクステント，すなわち動的スコープを持っています．すなわちその変数が作られた関数および，そこから呼び出されている関数からは参照できますが，その関数の外からは参照できません(Fig. 3.3)．

```
(f 1 2)
   .........

(de f (x y)
   .........
   (g 3 4)
   .........
)

(de g (p q)
     .........
     (h 5)
     .........
)

(de h (z)
     .........
)
```

関数 f が呼ばれた

関数 f の中で関数 g を呼んでいる

関数 g の中で関数 h を呼んでいる

関数 g の中からは，自分のローカル変数 p，q に加えて x，y も参照できる

関数 h の中からは，自分のローカル変数 z に加えて，x，y，p，q も参照できる

関数hが呼ばれた時点の環境リスト

((z . 5) (q . 4) (p . 3) (y . 2) (x . 1).........)

この状態で x を参照すれば，当然 1 という値がbindされている

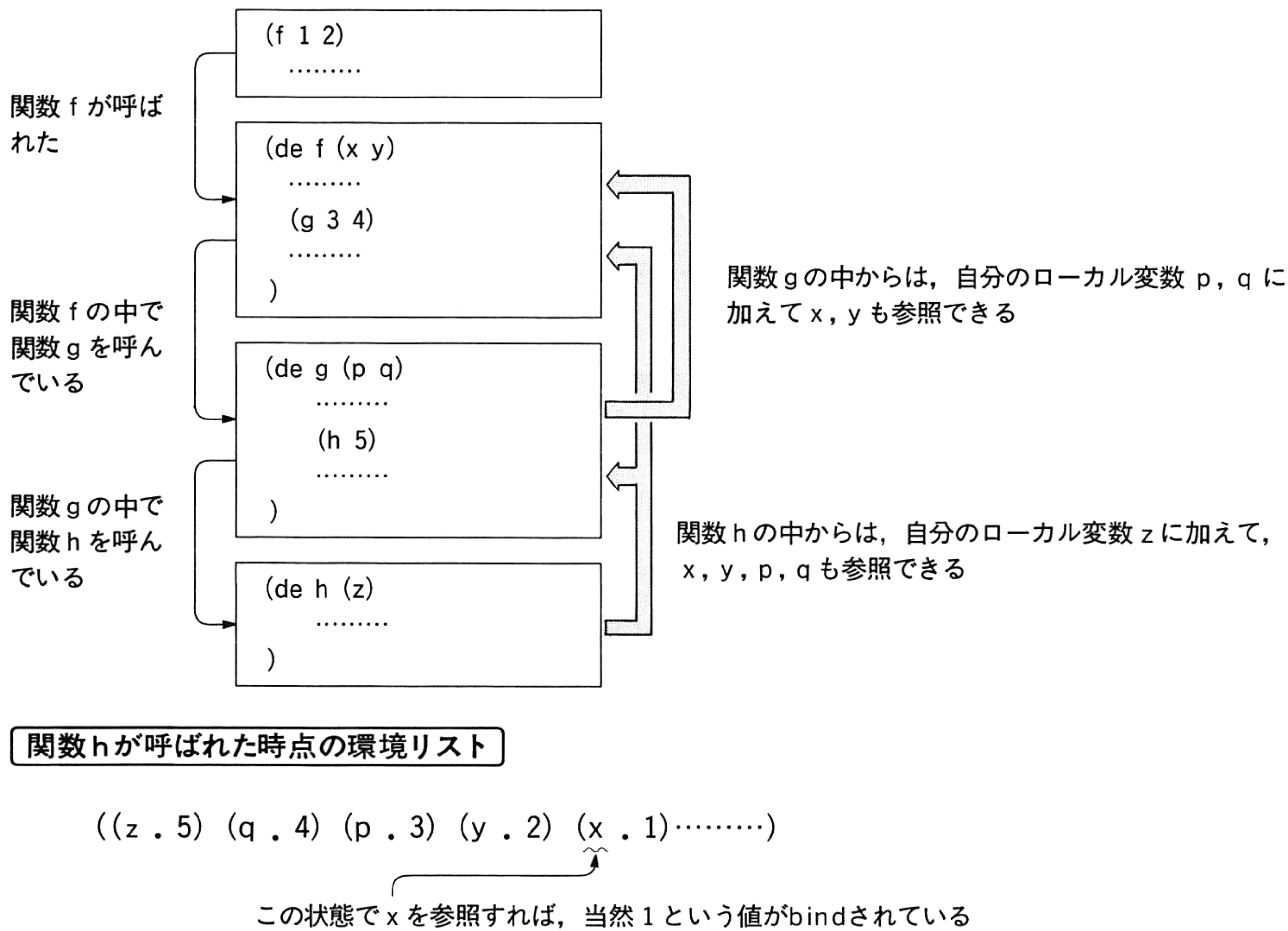



Fig. 3.3　動的スコープ

これに対し，lambda bind されていない普通のアトムは，無限スコープと無限エクステントを持っています．つまり，いつ，どこからでも参照できるということです．

3.3.2 シャドウイング

Will o'Lisp のアトムは，グローバル変数としても，ローカル変数としても，無限スコープを取りますが，ある場合には，その値は見えなくなります．たとえば，あるアトムが lambda bind されれば，もうそのアトムのグローバル変数としての値は見えなくなることは，先に述べたとおりです．また，ある関数で lambda bind されたアトムが，その lambda 式の内側（lambda 式や関数の中，そこから呼ばれている関数の中，およびさらにそこから呼ばれる関数の中など）で，再び lambda bind されると，外側の関数のローカル変数としての値は，内側の関数の内側からは参照できません（Fig. 3.4）．この時，内側の関数の中からは外側の値が新しい lambda binding の影（shadow）に隠れて見えない，という意味で，このことをシャドウイングと呼んでいます．

```
(f 1 2)
    .........

(de f ([x] y)
    .........
    (g 3 4)
    .........
)

(de g (p q)
    .........
    (h 5)
    .........
)

(de h ([x])
    .........
)
```

関数 h の中で x というローカル変数を用いてしまったため，関数 f の中の x は見えなくなってしまう

関数 h が呼ばれた時点の環境リスト

これが見えなくなる

```
((x . 5) (q . 4) (p . 3) (y . 2) (x . 1).........)
```

この状態で x を参照すると，新しい (x . 5) という bind のため，元の (x . 1) という bind は隠されてしまう

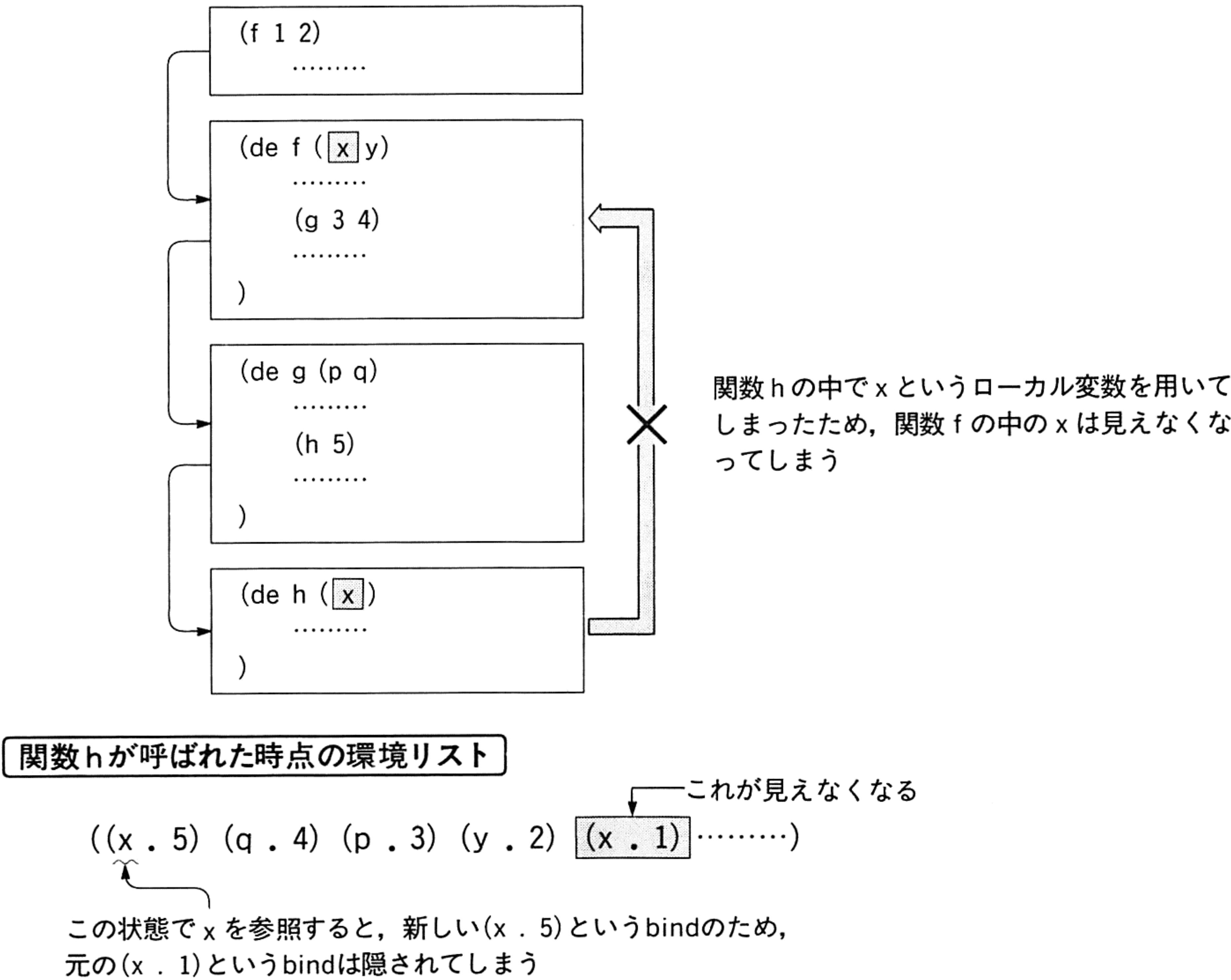



Fig. 3.4 シャドウイング

ローカル変数が無限スコープを持つ Lisp の場合，ある関数の中にある変数が，その関数のローカル変数でないならば，その変数がどんな値を参照するかは，その関数が置かれた場所に依存することになります．トップレベルで使われた場合は，グローバル変数として働くことになり，その変数を lambda bind している関数の内側で使われる時は，lambda binding による値が参照さ

れます．ローカル変数の意味の文脈依存性は，へたに使用するとプログラムを極めて理解しがたいものにする危険をはらんでいます．したがってある関数のローカル変数を下位の関数で参照したければ，引数として渡して変数の意味をはっきりさせておく方がよいといわれています．

3.3.3 deep binding と shallow binding

アトムの値の保持の方法は，環境リストの形で持つ方法と，メモリ上のスタックを使った方法の2つに大別されます．環境リストとは，アトムとその値のドット対のリストで，これを使う方法をdeep binding(深い束縛)といい，それに対しスタックを使う方法をshallow binding(浅い束縛)といいます．

deep bindingでは，アトムに値をbindする時には，アトムと値をドット対にして環境リストの先頭に*cons*します．アトムの古い値は環境リストの後ろの方にそのまま残っていますが，アトムの値を求める時には，環境リストの先頭から探し始めるので新しい値が先に見つかることになります．古い値を回復する時は(関数の適用を終えて仮引数の値を元に戻す時など)環境リストからアトムと新しい値のドット対を削除します．

shallow bindingでは，アトムの現在の値は常にアトムの中にあり，値を参照する時はアトムの中を見るだけですみます．setqなどで値を代入する時は直接アトムの中を書き変えて，仮引数に実引数を代入する時はスタックにアトムと古い値をプッシュして新しい値をアトムに格納し，関数の適用が済んだら古い値をポップして値を回復します(Fig. 3.5)．

一般に値の参照，更新は，shallow bindingの方が環境リストを検索しないですむ分，処理が速く簡単になります．LISP1.5は，deep bindingでしたが，Lispが普及するにつれ，しだいにより速いLispを求められるようになり，shallow bindingが支配的になりました．

一方deep bindingは，環境がリストで表されているのでLispで環境を操作することができるのが強みです．また，Lispコンパイラを使おうとすると，コンパイルされた関数では，引数がCPUのレジスタに置かれるため，必然的に静的スコープをとることになり，コンパイラを重視した場合，インタープリタも静的スコープをとった方が整合がとれて便利です．Common Lispが静的スコープをとっているのもそのためです．ところが，静的スコープと動的スコープを実現する手間はdeep bindingではたいして変わりませんが，shallow bindingで静的スコープを実現しようとすると，複数のメモリスタックを動的に管理しなければなりません．この困難を避けるため，どうせコンパイラを使うのならば，インタープリタの効率は多少落ちてもよかろうということで，deep bindingを採用するLispが，再び勢力を取り戻す可能性もあります．Will o'Lispはdeep bindingになっていますが，グローバル変数の値をアトムに置いた点でshallowに日和っています．

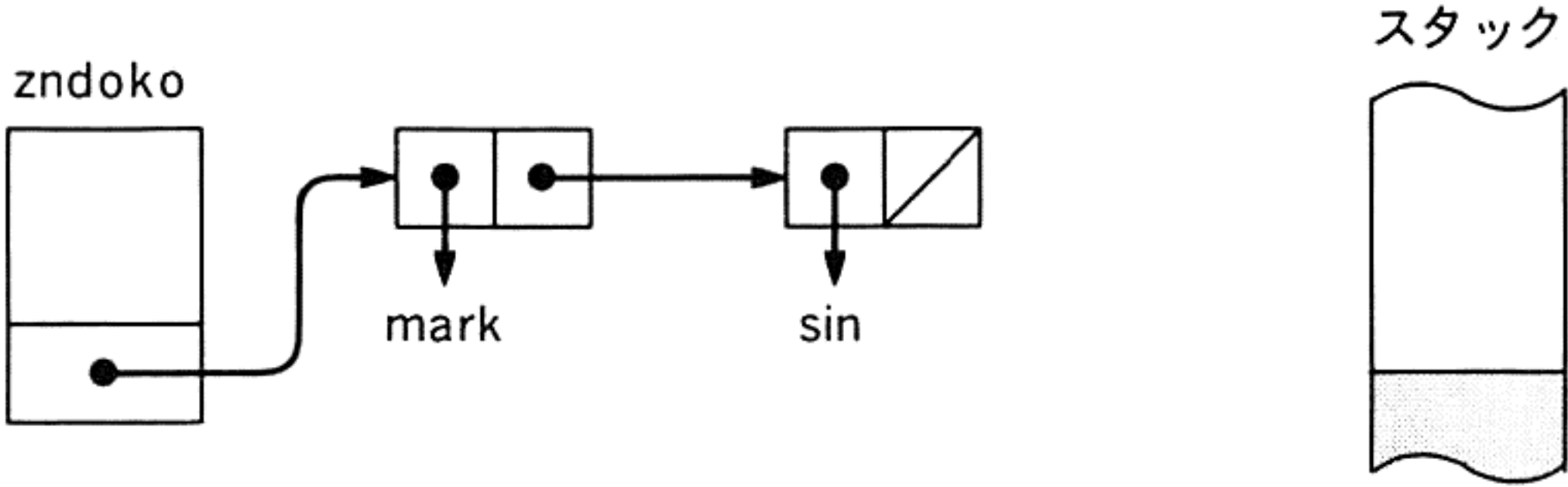
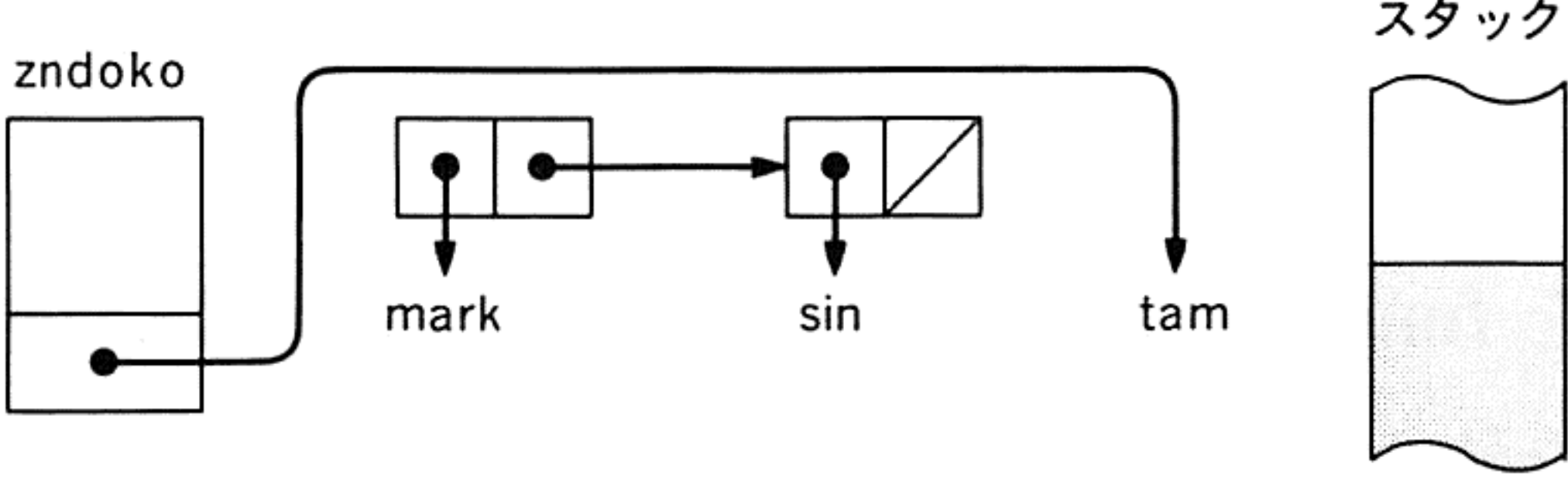
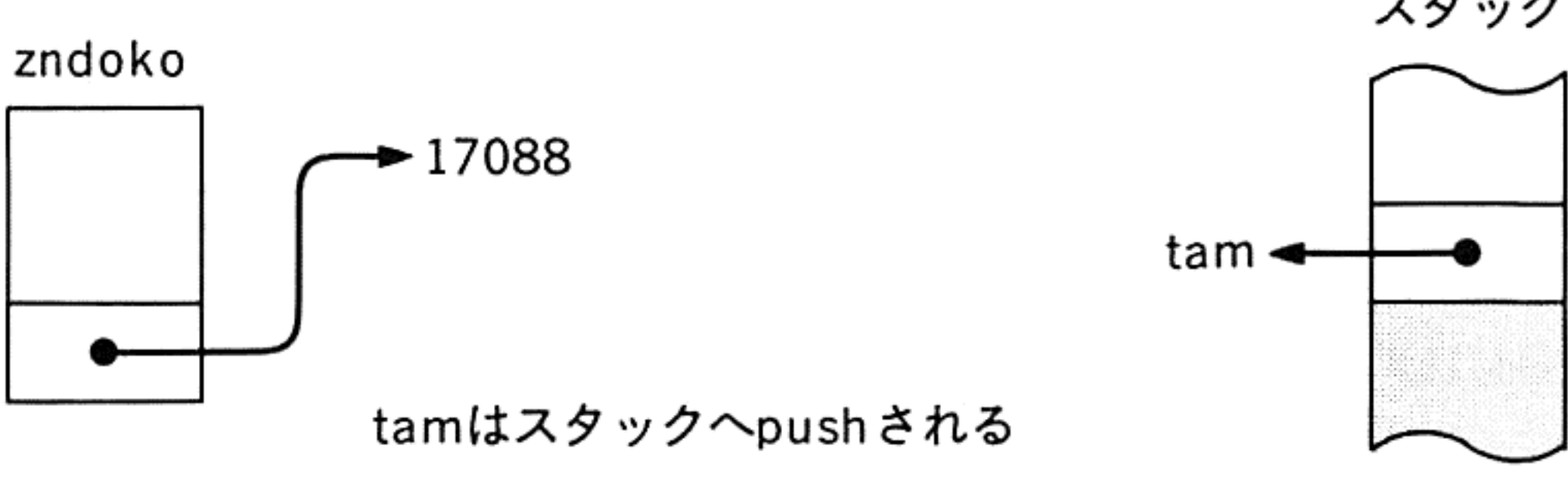
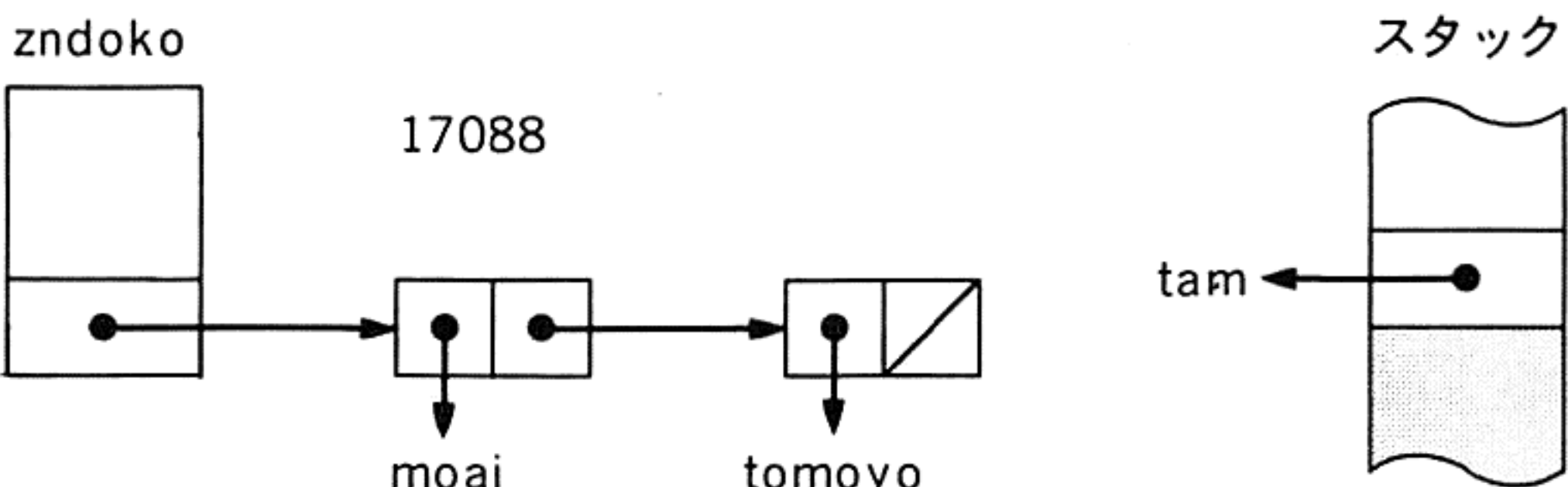
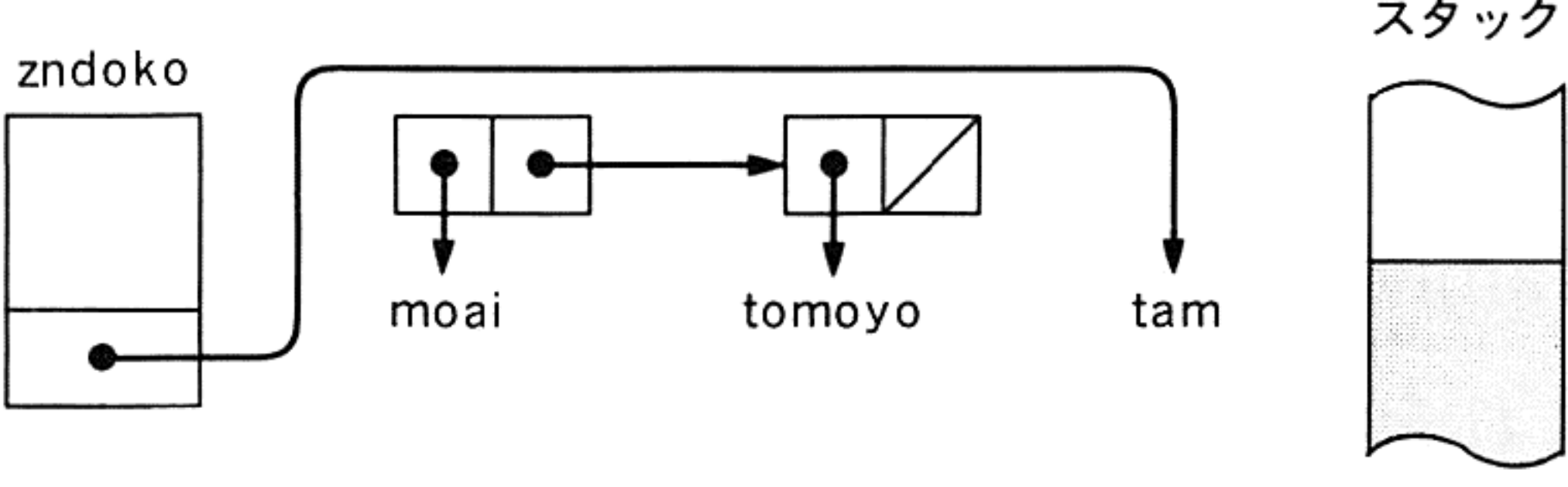
	deep binding
①アトムzndokoに値(mark sin)を代入	環境リスト：((zndoko . (mark sin)))
②アトムzndokoに別の値tamを代入	環境リスト：((zndoko . tam))
③zndokoを仮引数として実引数17088をlambda bind	環境リスト：((zndoko . 17088) (zndoko . tam))
④zndokoの値を参照する	環境リストが先頭から調べられ，最初に見つかった値が参照されるので，zndokoの値は17088になる
⑤仮引数zndokoに別の値(moai tomoyo)を代入	環境リスト：((zndoko . (moai tomoyo)) (zndoko . tam))
⑥lambda bindを解消	環境リスト：((zndoko . tam))

	shallow binding
①アトムzndokoに値(mark sin)を代入	zndoko mark sin スタック
②アトムzndokoに別の値tamを代入	zndoko mark sin tam スタック
③zndokoを仮引数として実引数17088をlambda bind	zndoko 17088 tam スタック tamはスタックへpushされる
④zndokoの値を参照する	アトムzndokoの中の値へのポインタが参照されるので，zndokoの値は17088になる
⑤仮引数zndokoに別の値(moai tomoyo)を代入	zndoko 17088 moai tomoyo tam スタック
⑥lambda bindを解消	zndoko moai tomoyo tam スタック

Fig. 3.5　deep binding と shallow binding

3.3.4 function式によるクロージャ

Common Lisp仕様書には，〝変数束縛は通常，静的スコープと無限エクステントを持つ〟と書いてあります．〝変数束縛〟とは，ほぼlambda bindingのことで，したがってこれはちょうどCの関数内で宣言された記憶クラスstaticのローカル変数と同様のふるまいを意味します．スコープに関しては，CommonとWill o'Lispははっきり異なっています．Commonの変数の意味はWill o'Lispと違って場所に依存しません．一方，エクステントの方は，先にWill o'Lispのローカル変数は動的エクステントを持つと述べましたが，実はCommonとそう違うわけではないのです．

Commonでは，lambda式を単独で使うことは考えておらず，常にその前に〝#'〟という記号をつけて使うことを想定しています．

#'(lambda ……)

は，次のような形式の略記です．

(function (lambda ……))

この形式をfunction式と呼びます．function式は，関数が定義される時の環境(どのシンボルアトムがどんな値を持ち，どんな関数を表しているか，など，何がどう定義されているかに関する情報)を保存するために使います．function式を評価すると，その中のlambda式に関するクロージャ(closure：閉包)が評価値となります．クロージャとは，関数とその環境をセットにしたもので，単純なlambda式の代わりにfunction式を使うと，その中のlambda式が実行される時に，クロージャの中の環境で評価がおこなわれ，その中のlambda式の実行が終了しても，lambda bindingされた変数の値はクロージャの環境中に保存され，次回そのlambda式が実行される時再び参照できます．Common Lisp仕様書は環境の保存方法を特に規定していません．Will o'では，LISP1.5にならい，funarg形式でクロージャを実現しています．

Will o'Lispでは，function式を評価すると，次のようなfunarg形式が返されます．

(funarg ＜lambda式＞ ＜環境リスト＞)

この中のlambda式はfunction式中のそれで，環境リストは，function式が評価される時点のものです．これを先頭要素とする関数形式を評価する時には，Will o'Lispインタープリタは環境リストを取り出し，その環境の下でlambda式を評価するようにして，上記のクロージャを実現しています．したがって，Will o'Lispでもlambda式にかならずfunctionをつけて使うならば，そのlambda bindingは無限エクステントを持つことになります．

しかし，無限エクステントを持つといっても，ある時点の評価で仮引数やローカル変数に与えられた値が，その次の回の評価の時に参照できるとは限りません．というのは，仮引数ならば次回の評価に入る前に新しい実引数と bind されますし，ローカル変数でも次回の評価で宣言部を通れば初期化されてしまうからです．したがって，ある関数についてCの記憶クラス static のローカル変数のように働く変数は，その関数の外側で bind されている変数だけです．

以下の例を見てください．

```
% (de learn () (print memory) (setq memory spell))
learn
```

learn はアトム memory の値を表示した後，memory にアトム spell の値を代入する関数で，返す値も spell の値です．

```
% (setq troll
%       #'(lambda  (spell)
%              (let  ((memory none))   ……関数の中にローカル変数 memory を持たせる
%                    (learn))))
(funarg (lambda (spell) (let ((memory none)) (learn)) nil)
```

関数 *let* はローカル変数を使うための関数です．*let* の第1引数は，ローカル変数宣言で，

((memory none))

と書くと，ローカル変数 memory が初期値 none と lambda bind され，その環境の下で残りの引数が評価され，最後の引数の評価値が *let* の値になります．ここではアトム troll に，spell を引数にとって関数 *learn* を呼ぶ関数を代入しています．関数は function 式の形で与えられ，それが評価された結果，funarg 式が返されています．funarg 式の最後に環境（ここでは何も lambda bind されていないので nil になっている）が見えています．

```
% (let ((memory none))   ……関数定義の外側で memory に "none" を bind し，
%       (setq ogre_lord                    ((memory .none))という環境を作る
%              #'(lambda (spell) (learn))))
(funarg (lambda (spell) (learn)) ((memory . none)))
```

アトム ogre_lord にも同様な関数が代入されていますが，代入が *let* の内側でおこなわれているところが違います．そのため，*let* による lambda bind の結果が環境として funarg 式の最後に見えています．

```
% (let ((memory none))
%         (setq fire_giant
%                '(lambda (spell) (learn))))
(lambda (spell) (learn))
```

アトム fire_giant に代入されている関数は, function 式ではなく, ただのクォートのついた lambda 式になっています.

```
% (funcall troll 'lahalito)   ……memory をローカル変数として持つ関数 troll に〝lahalito〞を与えて
none                              覚えさせる
lahalito

% (funcall troll 'lahalito)   ……もういちど troll を実行してみると, 前回覚えた memory の内容は消
none                              えている
lahalito
```

関数 *funcall* は, 第1引数の関数に, 残りの引数を与えて実行する関数です. 普通の関数作用と違うのは, 関数作用では, 関数であるリストの第1要素は評価されないのに対し, funcall では, 第1引数を評価したものを関数として扱っていることです.

troll に lahalito を与えると, troll はまず memory を none に初期化してから, lahalito と比べるので, せっかく1回目で memory に lahalito を代入してもその結果は残りません.

```
% (funcall ogre_lord 'lahalito)   ……memory を関数クロージャの中に持つ関数 ogre_lord に
none                                  〝lahalit〞 を与えて覚えさせる
lahalito

% (funcall ogre_lord 'lahalito)   ……もういちど ogre_lord を実行してみると, 前回覚えた memory
lahalito                              の内容が残っている
lahalito

% memory                          ……グローバル変数 memory の内容には手をつけていない
memory
```

ogre_lord では, 関数の中では初期化はおこなわれないので, 1回目の実行で memory に代入された値は, 2回目の実行の最中に参照できます.

```
% (funcall fire_giant 'lahalito)   ……memory を持たない関数 fire_giant に 〝lahalito〞 を与えて
memory                                 覚させる
lahalito
```

```
% (funcall fire_giant 'lahalito)    ……もういちど fire_giant を実行してみると，確かに情報は記
lahalito                               憶されている
lahalito

% memory                             ……しかし，それはグローバル変数を書き換えてしまうもので
lahalito                               あった
```

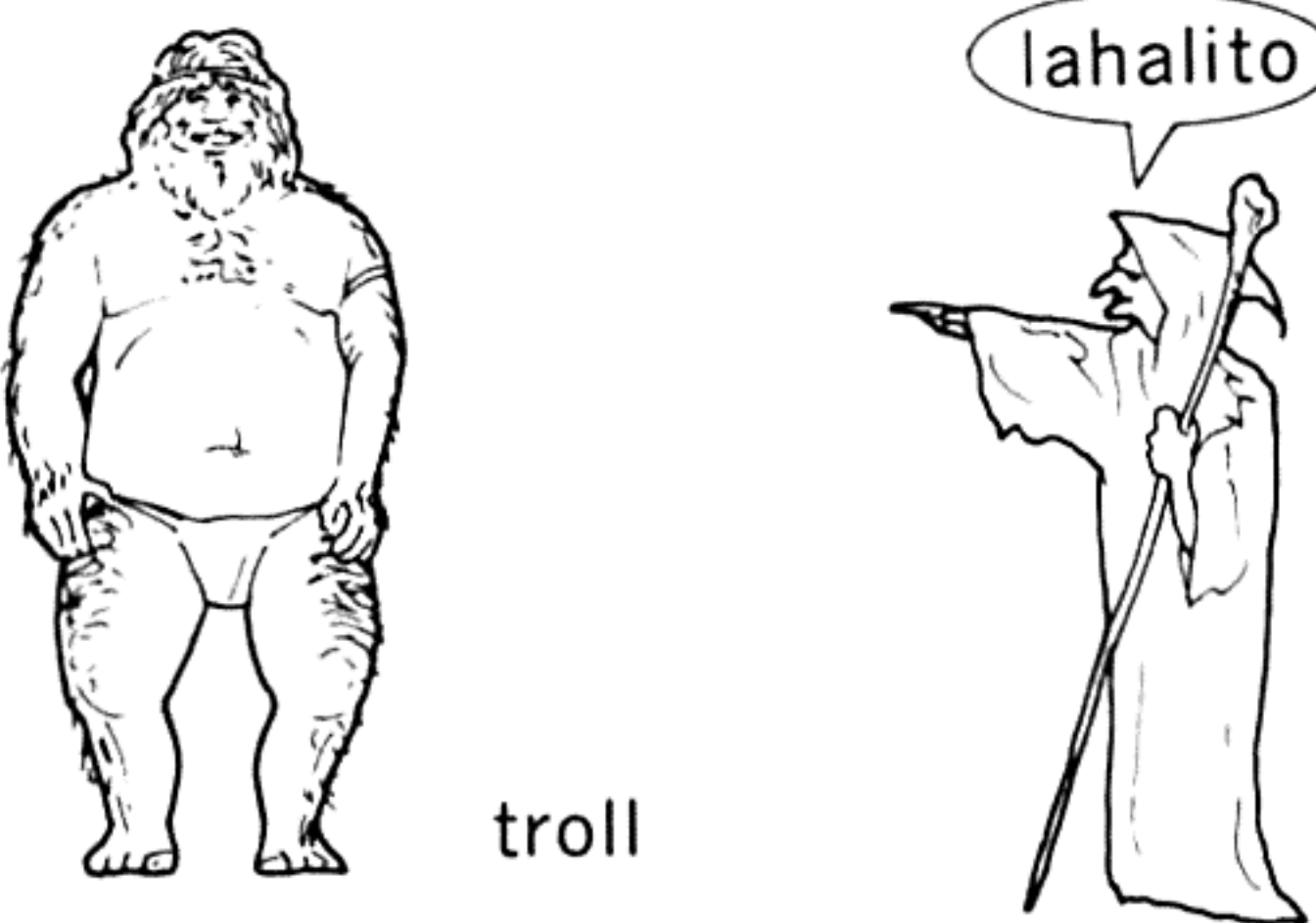
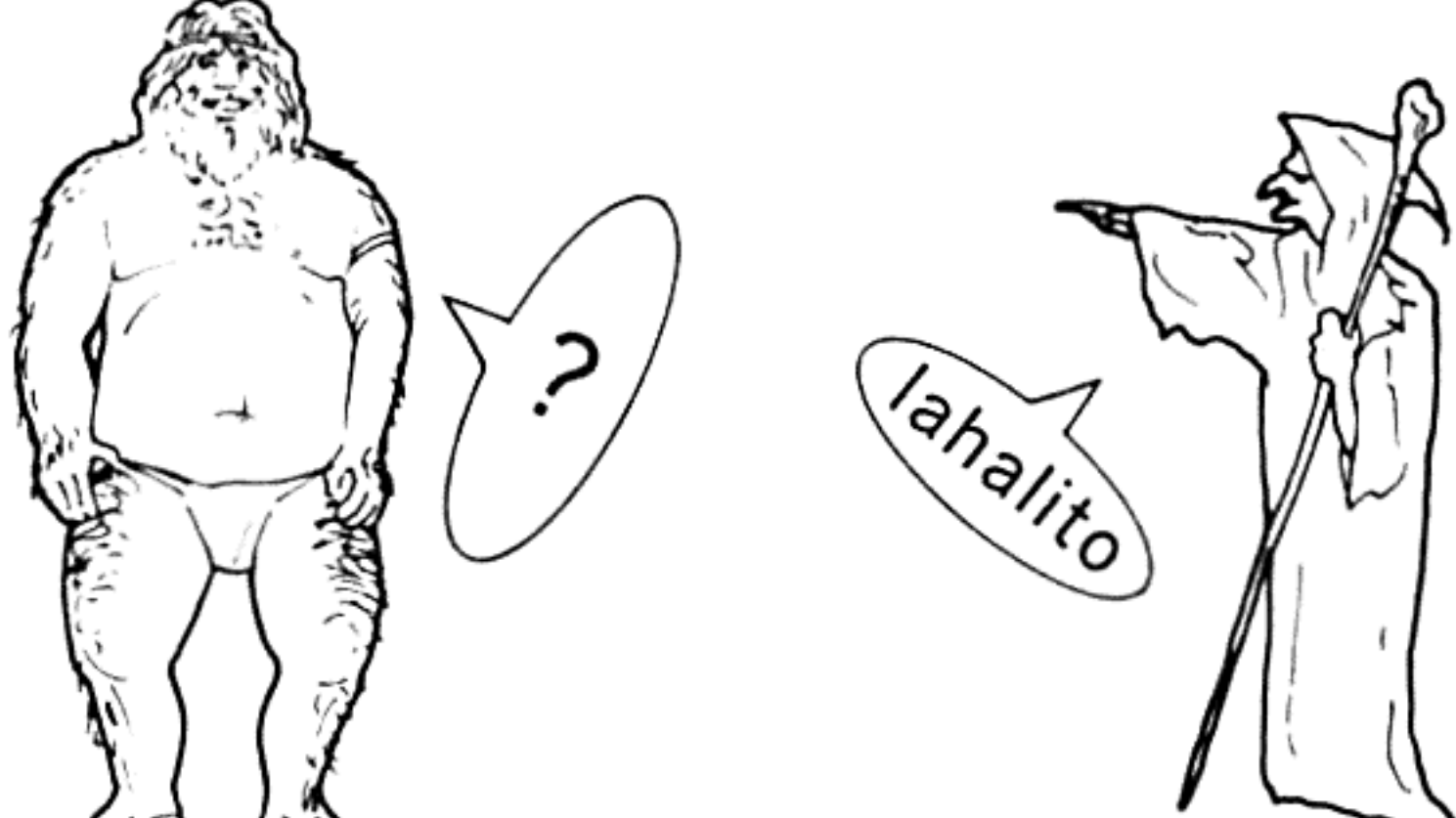
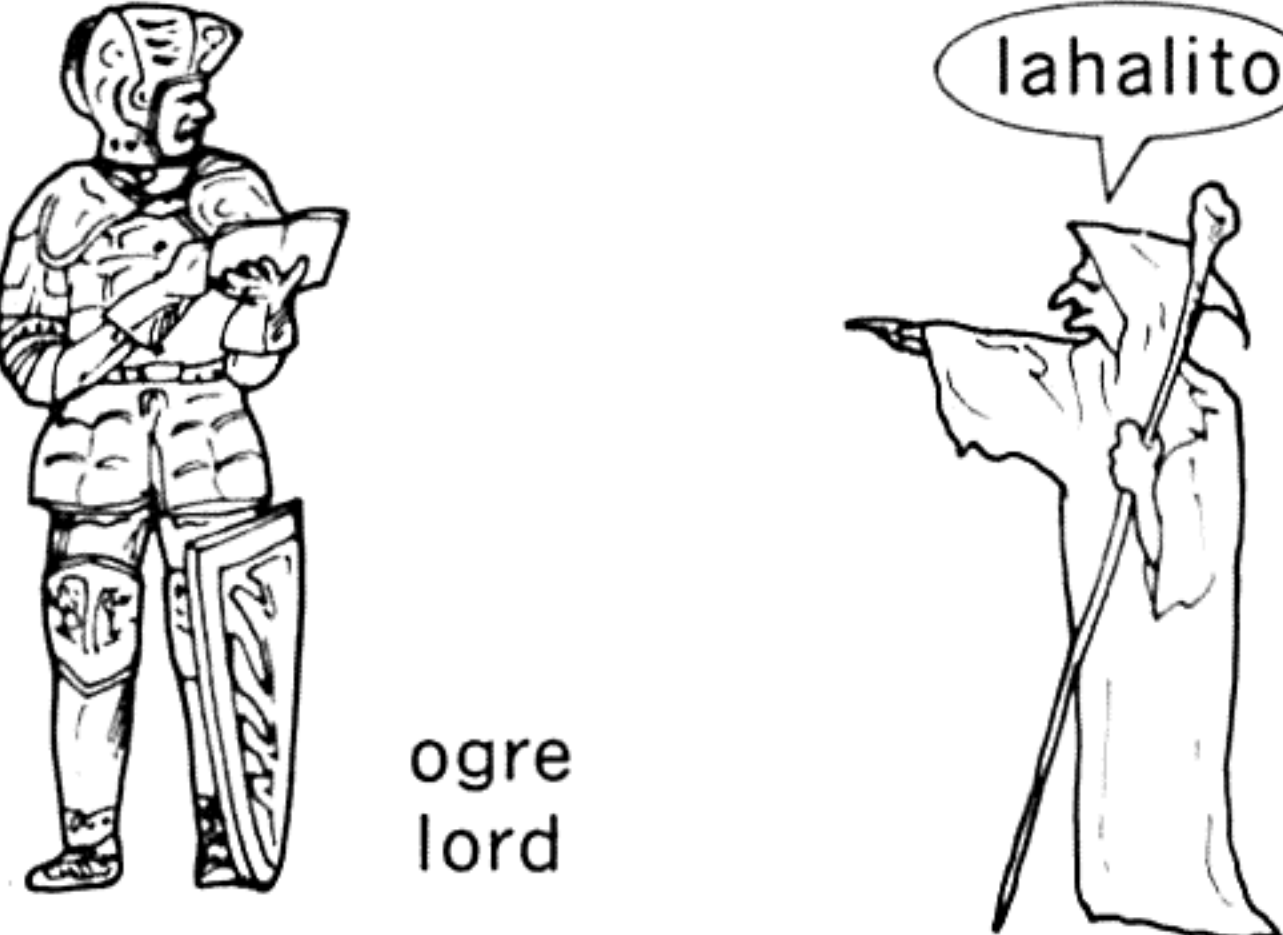
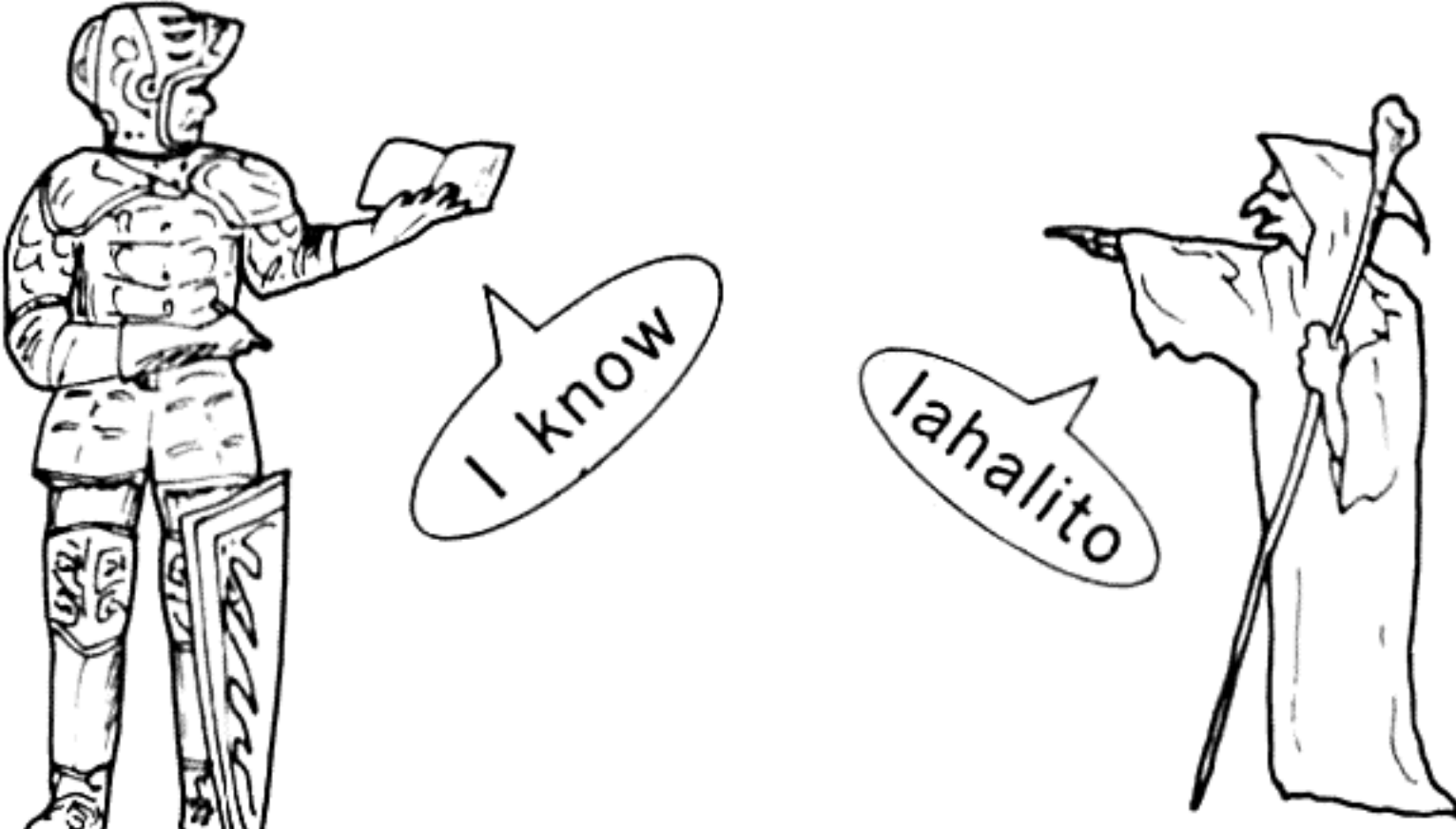
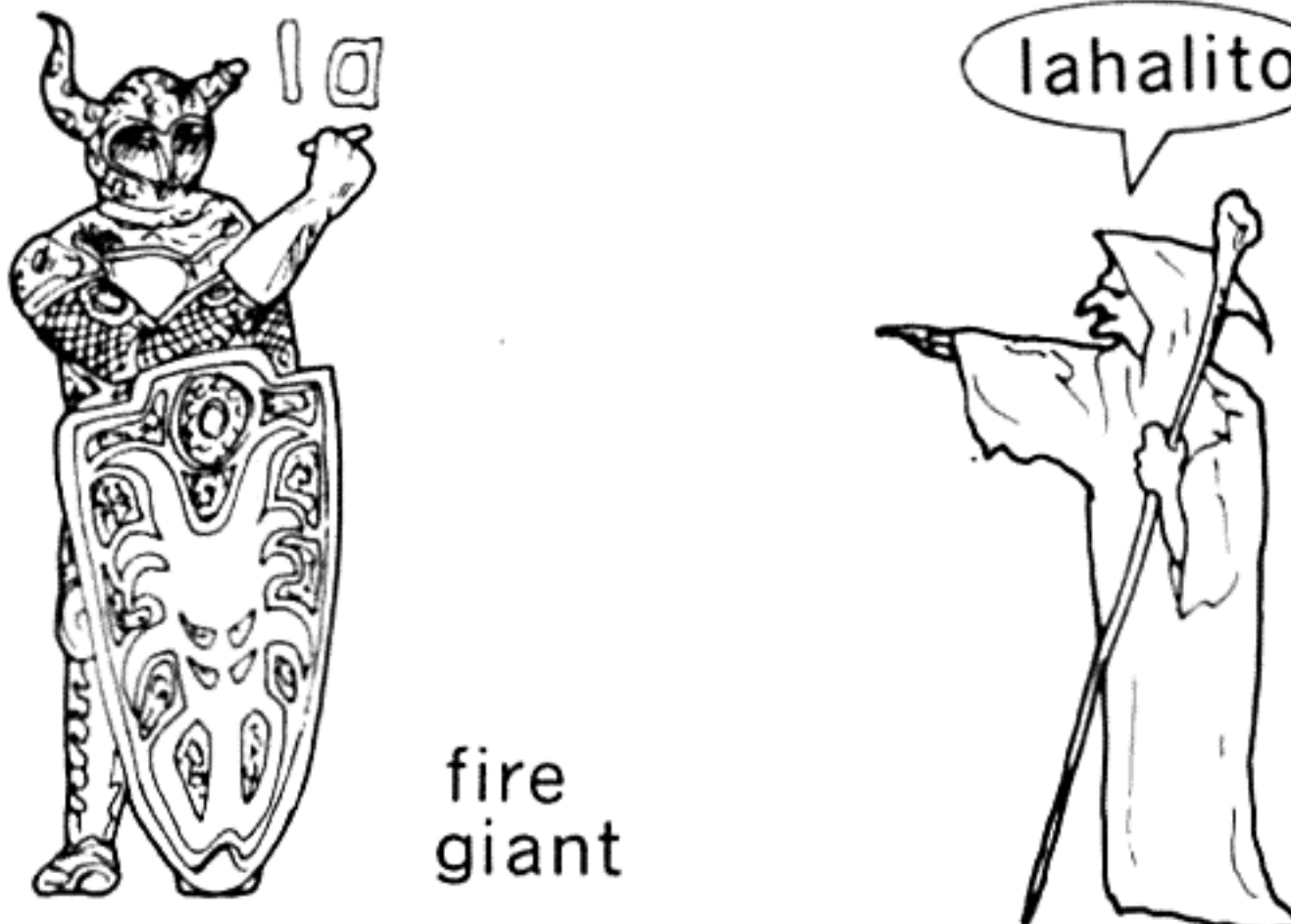
いまの 2 つの関数では memory はローカル変数だったので，アトムの中の値は，変わっていません．他で使っていなければ，オートクォートによる memory のアトム中の値 memory がそのまま残っているはずです．ところが，fire_giant の関数の場合，ただの lambda 式なので，環境が保存されていません．それがアトム fire_giant と bind される時には，アトム memory はアトム none と lambda bind されていますが，*let* を抜けてしまうとその lambda bind は解消されるので，トップレベルで使うと memory はローカル変数ではなくグローバル変数になります．したがって lahalito を与えると，値は環境リストではなく，アトムの中に直接置かれることになり，次回の実行時でもそのまま残っています(Fig. 3.6)．

	1 回 目	2 回 目
memoryをローカル変数として持つ関数troll	lahalito troll	memoryは関数の評価が終わるごとに消滅してしまう ? lahalito
memoryをクロージャの中に持つ関数ogre_lord	lahalito ogre lord	memoryは次の評価の時にも残っている I know lahalito
memoryを持たない関数fire_giant	la lahalito fire giant	グローバル変数memoryを書き換えてしまう lahalito I know lahalito

Fig. 3.6 関数クロージャ

3.4 funarg問題

関数の環境の保存は，特に汎関数(functional)を使う場合に重要となります．汎関数とは関数を引数とする関数のことで，引数となっている関数を関数引数(function argument)といいます．たとえば，次の関数を見てください．

```
% (de filter (x y)
%         (cond  ((null y) nil)
%                ((funcall x (car y)) (cons (car y) (filter x (cdr y))))
%                (t (filter x (cdr y)))))
filter

% (filter '(lambda (x) (greaterp x 5)) '(3 8 2 1 7 5))     ……5より大きな数値のみ抜き
(8 7)                                                         出す

% (filter '(lambda (x) (greaterp (alen x) 3))            ……3文字より長い名前のアト
%          '(car cdr cond atom eq))                          ムのみ抜き出す
(cond atom)
```

関数*filter*は，述語関数とリストを引数にとり，リストの要素のうち，述語関数がnilでないもののみをリストにまとめたものを返します．さて，これを使って敷居値と数値アトムからなるリストをとり，そのうち敷居値より小さいものだけからなるリストを返す関数を定義します．

```
% (de lessthan (y z)
%         (filter '(lambda (x) (lessp x y)) z))
lessthan

% (lessthan 4 '(1 9 4 2))

Oops !
Error No.15 Illegal arguement -- Number required.
At (lessp x y)
```

するとこのようにエラーとなってしまいます．このエラーは，関数*lessthan*の中の関数引数が適用される時の環境が，*lessthan*の定義中にこのlambda式が見られる位置での環境と異なっていることにより生じたものです(それぞれを関数引数の適用環境，定義環境と呼ぶ)．これはfunarg問題と呼ばれるものの一種です．

何が起きているのかを調べるために，Lispになったつもりで関数を実行していくことにします．まず，評価するべきＳ式は，

```
①    (lessthan 4 '(1 9 4 2))
```

です．実引数を評価すると，

```
②    4  と  (1 9 4 2)
```

になります．*lessthan*の定義は，

```
③    (lambda (y z) (filter '(lambda (x) (lessp x y)) z))
```

です．仮引数は

```
④    (y z)
```

です．②と④をbindすると，定義環境⑤ができます．

```
⑤    ((z . (1 9 4 2)) (y . 4))
```

この環境の下で，③の本体，

```
⑥    (filter '(lambda (x) (lessp x y)) z)
```

を評価します．実引数を評価すると，

```
⑦    (lambda (x) (lessp x y))  と  (1 9 4 2)
```

になります．*filter*の定義は，

```
⑧    (lamdba (x y)
        cond ((null y) nil)
             ((funcall x (car y)) (cons (car y) (filter x (cdr y))))
             (t (filter x (cdr y)))))
```

です．仮引数は，

```
⑨    (x y)
```

です．⑦と⑨を⑤の上にbindすると，適用環境⑩ができます．

```
⑩    ((y . (1 9 4 2)) (x . (lambda (x) (lessp x y))) (z . (1 9 4 2)) (y . 4))
```

ここで,問題なのが,⑩の中にyが2回現れていることです.この環境の下ではyの値は(1 9 4 2)になります.そこで,この環境の下で,⑧の本体

⑪
```
(cond ((null y) nil)
      ((funcall x (car y)) (cons (car y) (filter x (cdr y))))
      (t (filter x (cdr y)))))
```

を評価すると,*cond* の最初の cond 節の条件部

⑫ (null y)

は nil になるので,次の cond 節の条件部

⑬ (funcall x (car y))

を評価します.*funcall* の引数を評価すると,x は

⑭ (lambda (x) (lessp x y))

になり,(car y)は

⑮ 1

になります.*funcall* は⑭を⑮に適用します.すると,⑭の仮引数

⑯ (x)

と⑮が bind されて,実行環境

⑰ ((x . 1) (y . (1 9 4 2)) (x . (lambda (x) (lessp x y))) (z . (1 9 4 2)) (y . 4))

ができます.この環境の下で⑭の本体

⑱ (lessp x y)

を評価します.実引数を評価すると,

⑲ 1 と (1 9 4 2)

になります. *lessp* は引数に数値アトムを必要とする関数なのに,⑲でリストが実引数として与えられたため,上記のエラーが起きていたのです!

この問題は表面的には，*lessthan* と *filter* が同じ仮引数 y を使っていることが原因です．したがってどちらかの仮引数の名前を変えればエラーは出ません．しかし，それでは *filter* を使うためには，同じ名前を使わないように，*filter* の仮引数の名前を覚えておかなければなりません．この問題は定義環境が保存され，適用環境として使われれば本質的に解決します．これは function 式と funarg 式の働きそのものです．実は function 式はまさにこの目的のために作られたものです．これを使って *lessthan* を書き直してみます．

```
% (de lessthan (y z)
%       (filter #'(lambda (x) (lessp x y)) z))
lessthan
```

前の定義と違っているのは lambda 式の前に “'” のかわりに “#'” がついたことだけです．しかし，これにより適切な環境の下で関数引数が実行されるようになります．これも評価の各段階を追いかけてみましょう．

定義環境ができるところまでは先ほどとほぼ同じです．ただし，*lessthan* の定義は，今度は，

⑳　(lambda (y z) (filter #'(lambda (x) (lessp x y)) z))

です．定義環境はさっきと同じで，

⑤　((z . (1 9 4 2)) (y . 4))

です．この環境の下で，③の本体，

⑥　(filter #'(lambda (x) (lessp x y)) z)

を評価します．先ほどと決定的に違うのは，実引数を評価すると，

㉑　(funarg (lambda (x) (lessp x y)) ((z . (1 9 4 2)) (y . 4))と　(1 9 4 2)

になることです．funarg 式の中には⑤の定義環境が入っています．㉑は *filter* の仮引数

⑨　(x y)

と bind されて，次のような環境ができます．

```
㉒  ((y . (1 9 4 2)) (x . (funarg (lambda (x) (lessp x y))
                                   ((z . (1 9 4 2)) (y . 4)))
      (z . (1 9 4 2)) (y . 4))
```

filterの定義の本体

⑪　(cond ((null y) nil)
　　　((funcall x (car y)) (cons (car y) (filter x (cdr y))))
　　　(t (filter x (cdr y)))))

は環境㉒の下で評価されます. ⑪の *cond* の最初の cond 節は先ほどと同様に nil になり, 2つ目の cond 節の条件部

⑬　(funcall x (car y))

が評価されます. *funcall* の引数 x を評価すると, funarg 式

㉓　(funarg (lambda (x) (lessp x y)) ((z . (1 9 4 2)) (y . 4))

になり, funcall はこれを〝(car y)〃の評価値

⑮　1

に適用します. 先ほどと違うのは, この時, funarg 式㉓から環境

㉔　((z . (1 9 4 2)) (y . 4))

がとりだされ, これが適用環境となることです. ㉓の中の lambda 式

㉕　(lambda (x) (lessp x y))

は環境㉔の下で⑮に適用されます. ㉕の仮引数が⑮に bind されると,

㉖　((x . 1) (z . (1 9 4 2)) (y . 4))

という実行環境ができ, この環境の下で, ㉕の本体

⑱　(lessp x y)

を評価すれば, ⑱の仮引数 x, y はそれぞれ

㉗　1 と 4

となって, *lessp* に正しい引数が渡されるわけです.

以上のように, deep binding の Lisp では, すべての関数引数に〝'〃の代わりに〝#'〃をつけておけば, funarg 問題は生じません.

shallow bindingのLispでは，funarg式のようなクロージャを作るのが面倒なため，マクロを用いて解決します．この場合ならば，*filter*を次のようなマクロとします．

```
% (dm filter (x y)
%       (setq x (eval x)
%             y (eval y))
%       ^(cond ((null ',y) nil)
%               ((,x ',(car y)(cons ',(car y)(filter ',x ',(car y))))
%               (t (filter ',x ',(cdr y)))))
filter
```

この*filter*で〝(lessthan 4 '(1 9 4 2))″を評価すると，まず，先と同じ定義環境

⑤　((z . (1 9 4 2)) (y . 4))

ができます．この環境の下，

⑥　(filter '(lambda (x) (lessp x y)) z))

が評価されると，この関数作用のcdr 部が評価されずに*filter*の仮引数とbindされて，展開環境

```
㉘ ((y . Z)) (x . '(lambda (x) (lessp x y)))
    (z . (1 9 4 2)) (y . 4))
```

ができます．*filter*の定義の本体は，展開環境の下で順に評価されます．さて，ここからがマクロの特殊なところで，最後の本体の評価値(マクロ展開)である，

```
㉙ (cond ((null '(1 9 4 2)) nil)
         (((lambda (x) (lessp x y)) '1)
          (cons '1 (filter '(lambda (x) (lessp x y)) '(9 4 2))))
          (t (filter '(lambda (x) (lessp x y)) '(9 4 2))))
```

が，展開環境㉘ではなく，定義環境⑤の下で，もう1度評価されます．つまり，マクロ展開が⑥の代わりに置かれていたのと同じ結果が得られることになり，*lessp*の引数のyの値は期待どおり4になってくれます．

shallow bindingのLispでは，funarg式のようなクロージャを作るのが面倒なため，マクロを用いて解決します．この場合ならば，*filter*を次のようなマクロとします．

```
% (dm filter (x y)
%        (setq x (eval x)
%              y (eval y))
%       ^(cond ((null ',y) nil)
%               ((,x ',(car y)(cons ',(car y)(filter ',x ',(car y))))
%               (t (filter ',x ',(cdr y)))))
filter
```

この*filter*で〝(lessthan 4 '(1 9 4 2))″を評価すると，まず，先と同じ定義環境

⑤　((z . (1 9 4 2)) (y . 4))

ができます．この環境の下，

⑥　(filter '(lambda (x) (lessp x y)) z))

が評価されると，この関数作用のcdr 部が評価されずに*filter*の仮引数とbindされて，展開環境

```
㉘　((y . Z)) (x . '(lambda (x) (lessp x y)))
     (z . (1 9 4 2)) (y . 4))
```

ができます．*filter*の定義の本体は，展開環境の下で順に評価されます．さて，ここからがマクロの特殊なところで，最後の本体の評価値(マクロ展開)である，

```
㉙　(cond ((null '(1 9 4 2)) nil)
          (((lambda (x) (lessp x y)) '1)
           (cons '1 (filter '(lambda (x) (lessp x y)) '(9 4 2))))
           (t (filter '(lambda (x) (lessp x y)) '(9 4 2))))
```

が，展開環境㉘ではなく，定義環境⑤の下で，もう1度評価されます．つまり，マクロ展開が⑥の代わりに置かれていたのと同じ結果が得られることになり，*lessp*の引数のyの値は期待どおり4になってくれます．

4章　Will o'Lispの仕様

ここまで，Lisp という言語が持つさまざまな特徴を述べてきました．本章では，この Lisp 処理系を作成するために，さらに細かい仕様を決定します．

4.1 機能の決定

Lispという言語には，長いあいだ〝標準仕様〟が存在しませんでしたが，最近になって，やっとCommon Lispが登場し，Lisp界の混乱にも終止符が打たれようとしています．たいへんに喜ばしいことです．これから新しく作られるLisp処理系は，何らかの意味でCommonの仕様を無視することはできないでしょう．

本書で作成するLispも，できることならば，Common Lispを目指したいとは思います．しかし，Common Lispは非常に大きな規模の処理系であり，1冊の本の限られたページの中で，そのフルセットを作ることは不可能です．

そこで本書では，近代的Lispとして最低限必要と思われる機能のみを備えた処理系の作成をおこないます．

まず，Lispにとって基本的な機能はすべて備えるようにします．ここで〝基本的〟というのは，だいたいLISP1.5程度の機能を考えています．したがって，少なくとも，以下の条件はクリアするものとしましょう．

- ・ガベージコレクションを完璧におこなうこと．
- ・*prog*関数が使用できること．
- ・マップ関数が使用できること．
- ・expr型とfexpr型の関数が定義できること．
- ・cond式の条件節や関数定義の本体は〝implicit progn(暗黙の*progn*)〟が使用できること．
- ・クォート〝'〟によって，*quote*関数の略記ができること．

また，本書の目的のひとつは，読者の方々にLispというものを知っていただくことです．その意味では，まずLispに興味を持ってもらうために，最近のLisp処理系にとりいれられている〝面白そうな〟機能もできることなら備えたいものです．これは，たとえば次のような機能が挙げられます．

- ・*catch* & *throw*による大域脱出
- ・*format*関数による書式付き出力
- ・*let*関数によるローカル変数の使用
- ・macro型の関数の定義
- ・バッククォートによる見やすいリスト表記
- ・日本語(漢字)の処理

こうして, あれこれと考慮した結果, 決定したのが次の項で述べる Will o'Lisp の言語仕様です. ただし, 何度もいうようですが, 1 冊の書籍のページには限りがあります. その中に収めるために, やむなく削除したという機能も, 実は多々あります.

しかし, 本書に掲載したプログラムリストが理解できた方であれば, この Lisp の仕様を拡張することはたやすいでしょう. そのために必要な情報は, 現在の Will o'Lisp の中に, すべて含まれているはずです.

4.2 Will o'Lispの言語仕様

それでは, Will o'Lisp の言語仕様を順に述べていくことにしましょう.

4.2.1 起動法

Lisp 立上げ時に〝initial.lsp〞というファイルがカレントディレクトリの中にあると, Will o'Lisp は自動的にそれを読み込み, その中に記述された関数を評価します. このファイルの中で, よく使う関数などの定義をおこなっておくとよいでしょう.

4.2.2 トップレベルループのテンプレート機能

トップレベルの read-eval-print のループにおいて, つぎの 7 つのシンボルアトムは特別な値と常に束縛させられています. Lisp 立上げ時, これらにはすべて nil が束縛されています.

+	:	前回のループで入力された S 式
++	:	前々回のループで入力された S 式
+++	:	前々々回のループで入力された S 式
−	:	このループで入力されたばかりの S 式
*	:	前回のループで評価された結果の S 式
**	:	前々回のループで評価された結果の S 式
***	:	前々々回のループで評価された結果の S 式

なお, エラーが発生してトップレベルまで抜けたループはこの束縛には影響を与えません. つまり, そのループはなかったかのごとく扱われます.

```
% (plus 5 6 7)
18

% (quotient * 2)          ……* = 18
9

% (times ** 3)            ……** = 18
54

% (print ++)              ……++ = (quotient * 2)
(quotient * 2)
(quotient * 2)

% (list - -)              ……- = (list - -)
((list - -) (list - -))

% (car 'a)                ……故意にエラーを起こしてみる
Oops! ……

% (print +)
(list - -)                ……(car 'a)はエラーを起こしたので影響しない
(list - -)
```

4.2.3 リーダにおける略記その他

Will o'Lispでは，リーダによるS式の読み込み時に，いくつかの特別な記法を用いることが可能です．それらを以下に示します．

●略記

- 'form　(quote form)の略記

●シャープ・サイン・マクロ

- #'form　(function form)の略記
- #: symbol　oblistに登録されないシンボルアトムを作る
- #xhhhh　hhhhを16進数として読み込む(h：0-9，a-f)
- #oNNNN　NNNNを8進数として読み込む(N：0-7)
- #¥nnn　10進数nnnを文字コードとして持つシンボルアトムを作る

●バッククォート構文

- ^form　バッククォート構文を作る.バッククォート内では〝,″と〝,@″が使える.
- ,form 及び ,@form　リスト内に〝form″を評価して挿入し,〝,@form″はリスト内に〝form″を評価して展開する. 内部的には,〝^form″は (backquote form) に,〝,form″は (comma form) に,〝,@form″は (atmark form) へという具合に関数化されている.

●その他

- 〝¥″の後の文字は英字扱いされるので,これによりシンボルアトムの印字名に特殊文字やスペースを含ませることができる.
- 2つの〝|″に囲まれた中にある特殊文字は英字としての扱いを受ける.
- シンボルアトム名の末尾に〝¥″+CRがある時は,そのシンボルアトム名は次の入力行に継続する.
- 浮動小数点型の〝.″とドット対の〝.″では浮動小数点型のほうが優先する.

```
% #xff
255

% (setq x '(a b c))
(a b c)

% ^(x ,x ,@x)
(x (a b c) a b c)

% (princ 'Fire¥ Giant|, Ogre Lord|)
Fire Giant, Ogre Lord Fire¥ Giant¥,¥ Ogre¥ Lord

% 'To¥
% mas
Tomas

% '(2.a)
(2.000000 a)

% (princ #¥007)  ……beep音が鳴る
#¥007
```

4.2.4 関数の表記の方法

次節から，Will o'Lisp が備えている関数の機能と使用法を説明していきます．その際，各関数は次のような方法で表記することにします．

（<関数名> <引数 1> …… <引数 n>）
…… <機能> ……
…… <使用例> ……

<引数>の部分は，さらに引数の種類や個数を示すために，以下のような表記法を用います．

●引数の種類

引数は，次の記号で表します．必要に応じて，〝exp1〃，〝exp2〃のように添字を用いることや，〝value-exp〃 のように意味がわかりやすくなるような接頭語を付けることもあります．

数値	: num
シンボルアトム	: atm
リスト(nil を含む)	: lis
任意のS式	: exp
局所変数リスト	: vars
仮引数リスト	: args
実行部	: body

また，評価される位置の引数にはクォートを付けます．たとえば，

'num

という記号は，評価されて，その結果が数値になるような引数を表しています．

例：

(cons 'exp 'lis) ←関数 cons の引数は，第 1 引数，第 2 引数とも評価されてから cons に渡される
第 1 引数は任意の S 式だが，第 2 引数はリストでなくてはならない

●引数の個数について

決まった数の引数をとる関数は，その引数をすべて表記します．

例：

(putprop 'atm1 'exp 'atm2) ←関数 putprop は引数を 3 つとる

(reclaim) ←関数 reclaim は引数をとらない

省略してもよい引数は，角カッコで囲みます．

例：

(read ['num]) ←関数 read の引数は省略可能

不定個数の引数をとる関数は，その部分を〝{ 引数 ……}〟と表します．

例：

(list {'exp1 ……}) ←関数 list は 0 個以上の任意個の引数をとることができる

(append 'lis1 {'lis2 ……}) ←関数 append は 1 個以上の任意個の引数をとることができる

4.2.5 関数を定義する関数

Will o'Lisp の関数は，その評価の形式，および関数定義の与え方によって，Table 4.1 に示す 5 種類に分類されます．

関数のタイプ	記述の方法	評価の形式
subr	C で記述されている	引数を評価する
fsubr	C で記述されている	引数を評価しない
expr	lambda 式による定義	引数を評価する
fexpr	nlambda 式による定義	引数を評価しない
macro	macro 式による定義	引数を評価しない．結果を再評価する

Table 4.1 Will o'Lispの関数分類

これらの中で，expr 関数，fexpr 関数，macro 関数の 3 種類は，以下に述べる〝関数を定義するための関数〟によって定義されます．

● (de funcname-atm args {body1 ……})

expr 型関数を定義します．すなわち，funcname-atm で指定されたシンボルアトムの ftype を〝_EXPR〟とし，また，そのシンボルアトムの fptr に〝(lambda args {body1 ……})〟という関数定義を与えます．関数の実行部分である body は複数存在してもかまいません．その場合その関数を実行すると，body は前にあるものから順に評価され，最後に評価された body の評価値が，関数自体の評価値として返されます(〝implicit progn〟機能)．

● (df funcname-atm args {body1 ……})

fexpr 型の関数を定義します．すなわち，funcname-atm で指定されたシンボルアトムの ftype を "_FEXPR" とし，また，そのシンボルアトムの fptr に "(nlambda args {body1 ……})という関数定義を与えます．body が複数ある場合は，*de* と同様に実行時に implicit progn として評価されます．

● (dm funcname-atm args {body1 ……})

macro 型の関数を定義します．すなわち，funcname-atm で指定されたシンボルアトムの ftype を "_MACRO" とし，また，そのシンボルアトムの fptr に "(macro args {body1 ……})という関数定義を与えます．body が複数ある場合は，*de* と同様に実行時に implicit progn として評価されます．

上記の3種類の関数定義関数は，どれも "args" の位置の仮引数リストの形態を変えることで，決まった個数の引数をとる関数を定義することも，不定個数の引数をとる関数を定義することも可能です．以下にその方法を述べます．

●決まった個数の引数をとる関数の定義

仮引数リストを通常のリストの形にして関数を定義すると，決まった個数の引数を持つ関数の定義ができます．この時実引数は順に仮引数と bind されます．実引数が仮引数より足りなければ "Not enough arguments" エラーが発生します．実引数が余れば，余った分は無視されます．

```
% (de foo (x y) (format nil '|x=~a  y=~a~n|  x y))  …… 2つの引数を持つ関数
foo

% (foo 1 2)
x=1  y=2
nil

% (foo 1 2 3)  ……多すぎる引数は無視される
x=1  y=2
nil

% (foo 1)       ……引数が足りない場合はエラーとなる
Oops !
Error  No.10 : Not enough argments
```

●不定個数の引数をとる関数の定義

仮引数にアトムを指定すると，そのアトムに，すべての実引数をリストにまとめたものがbindされ，任意の個数の実引数を渡すことができます．

```
% (de goo x (format nil '|x=~a~n|  x))  ……任意個の引数を渡せる関数
goo

% (goo 1 2)
x=(1 2)
nil

% (goo 1 2 3)
x=(1 2 3)
nil

% (goo)  ……引数を与えないことも可能
x=nil
nil
```

● n個以上の不定個の引数をとる関数の定義

仮引数リストの最後にドット対がある場合には，通常のリストを仮引数リストとした場合同様，実引数が順に仮引数とbindされますが，もし仮引数の個数以上の実引数が渡されてきた場合，余った分の実引数も無視されず，すべてドット対の最後のcdrのアトムにbindされます．実引数が仮引数より足りなければ，〝Not enough arguments〞エラーが発生します．

```
% (de noo (x y . z) (format nil '|x=~a  y=~a  z=~a~n|  x y z))  ……
noo                                          2つ以上の任意個数の引数を持つ関数

% (noo 1 2)
x=1  y=2  z=nil
nil

% (noo 1 2 3)       ……2つ目以降の引数は，リストにまとめられて最後のアトムにbindされる
x=1  y=2  z=(3)
nil

% (noo 1 2 3 4)     ……同上
x=1  y=2  z=(3 4)
nil
```

● (getd 'funcname-atm)

シンボルアトム funcname-atm から関数定義を取り出します．

関数のタイプが subr 型，fsubr 型の場合は，関数タイプを表す数値(subr に対して 6，fsubr に対して 2)とスタートアドレスとのドット対を返します．

関数のタイプが expr 型，fexpr 型，macro 型ならば，それぞれの定義である lambda 式，nlambda 式，macro 式を返します．

● (putd 'funcname-atm 'def-exp)

シンボルアトム funcname-atm の関数定義部分に直接，def-exp で示される関数定義を書き込みます．def-exp は lambda 式，nlambda 式，macro 式，funarg 式の形のリストであるか，または関数 getd によって得られる subr 型や fsubr 型の関数定義でなければなりません．シンボルアトムの ftype には，def-exp にふさわしい関数タイプが自動的にセットされます．

```
% (putd 'second '(lambda (x) (car (cdr x))))   ……second に lamdba 式による関数定義を
(lambda (x) (car (cdr x)))                        与える

% (second '(1 2 3))
2

% (putd 'first (getd 'car))                     ……first に subr 関数 car の定義そのものを
(6 . 9987654200)                                  与える

% (first '(1 2 3))
1
```

● (function funcname-exp)

funcname-exp と現在の環境リストを組み合わせて，funarg 式にして返します．funcname-exp は，lambda 式，nlambda 式，macro 式の形のリストまたは関数名のシンボルアトムでなければなりません．通常は〝#'funcname-exp〟と略記します．

putd と funarg 式を併用することにより，関数クロージャが作成できます．関数クロージャとは，自分専用の環境を持つような関数であると考えてください．

```
% (de make-random (r)  ……乱数関数を作成する関数
%     #'(lambda () (setq r (add1 (times r 257)))))
make-random
```

```
% (putd 'rnd (make-random 1))  ……rndに，関数定義として，乱数の種rを持つような関数ク
(funarg (lambda nil (setq r (add1 (times r 257)))) ((r . 1)))   ロージャを与える

% (rnd)
258

% (rnd)
66307
```

4.2.6 シンボルアトムの値を代入する関数

● (setq atm1 'exp1 {atm2 'exp2 ……})

シンボルアトムatm1の値をexp1に，atm2の値をexp2に……と，順にセットします．atmが局所変数として環境リストに束縛されている時には，環境リスト中の値が書き換えられます．最後にセットされたexpの値が*setq*の評価値として返されます．

setqの引数の個数が奇数の場合はエラーになります．

t，nil，EOFの値を書き換えることは禁止されています．

● (psetq atm1 'exp1 {atm2 'exp2 ……})

*setq*と同様に，atmの値をexpにセットしますが，その操作に先立ってすべてのexpが評価されます．両者の違いを以下の例に示します．

```
% (setq x 'pin y 'tan)                     ……xにpin，yにtanを与える
tan

% (psetq x y y x)                          ……psetqによる並列代入操作
pin

% (format nil '|x=~a  y=~a~n| x y)        ……xとyの値を表示してみると，入れ替わっている
x=tan  y=pin
nil

% (setq x y y x)                           ……setqによる順次代入操作
pin

% (format nil '|x=~a  y=~a~n| x y)        ……xにyが代入され，そのxがyに代入されるため，
x=pin  y=pin                                 両者は同じになる
nil
```

● (set 'atm1 'exp1 {'atm2 'exp2 ……})

setqと同様に，シンボルアトムatm1の値をexp1に，atm2の値をexp2に……と，順にセットします．ただし，*setq*と異なり，atmの位置の引数も評価されます．

```
% (set (car '(zndoko pin)) 'tam)   ……(car '(zndoko pin))の評価値であるzndokoにtamが代
tam                                   入される

% zndoko
tam
```

4.2.7 シンボルアトムの属性を扱う関数

● (putprop 'atm 'value-exp 'propname-atm)

シンボルアトムatmに，propname-atmで指定される名前の属性と，その属性値としてvalueをあたえます．すでにpropname-atmという名前の属性が存在していれば，その属性値を書き換えます．与えられた属性値value-expを*putprop*の評価値として返します．

● (get 'atm 'propname-atm)

シンボルアトムatmからpropname-atmで指定される属性の属性値を取り出して返します．そのような属性が存在しなければnilを返します．

● (remprop 'atm 'propname-atm)

シンボルアトムatmからpropname-atmで指定される属性とその属性値を取り除き，取り除かれた属性値を返します．そのような属性が存在しなければnilを返します．

4.2.8 制御構造のための関数

● (cond {clause1 clause2 ……})

各clauseは

(<条件部> {body1 body2 ……})

の形を持ちます．各cluaseについて順に条件部が評価され，値がnilでなければ，body部分がimplicit prognとして実行され，最後に評価された値が*cond*の値として返されます．bodyがひとつもない場合には，条件部を評価した値が返されます．条件部の評価値がnilの時は，次のclauseについて同様のことをおこない，すべてのclauseの条件部がnilになっていれば，*cond*はnilを返します．

● (prog varlist {body1 body2 ……})

局所変数リスト varlist にしたがってシンボルアトムの binding をおこなった後, body が implicit progn として順に評価されます．ただし，アトムである body は，*go* 関数の飛び先を示すタグとして使用されるのみであり，評価はされません．

prog の中では，(return 'exp) と (go atm) が使用できます．(return 'exp) が評価されると，*prog* の実行は終了し，exp が最終的な *prog* の評価値として返されます．
(go atm) が評価されると，実行の流れは atm と等しい body の次に移り，そこから再び評価を続けます．

return 式が評価されずに終了すると，prog は nil を返します．

```
% (de square-sum x
%    (prog ((sum 0))
%        LOOP (cond ((null x) (return sum)))
%               (setq sum (plus sum (times (car x) (car x))))
%               (setq x (cdr x))
%               (go LOOP)))
square-sum

% (square-sum 1 2 3)
14
```

なお，*prog* の内側であれば，body としてあらわに現れていなくとも，*return*，*go* は使えます(すなわち，*prog* から呼ばれる関数の定義の中に唐突に *go*，*return* 式が現れることが状況により許される)．*prog* の内側に *prog* がネスティングしている状態で，内側の *prog* にないラベルにジャンプしようとした場合，外側の *prog* に脱出が起こり，外側の *prog* 内でラベルが検索されます．さらにそこでも見つからない時には，順次脱出が起こり，その脱出はトップレベルにまでおよびます．

● 局所変数リスト

局所変数リストは次の形を持ちます．

((atm1 'exp1) (atm2 'exp2) ……)

prog では，exp1，exp2，……は，すべてが評価されてから atm1，atm2，……に bind されます．(atm 'exp) の部分を単に atm と記述した場合には，atm には nil が bind されます．

● (progn {body1 body2 ……})

body を順番に評価し，最後に評価した値を返します．

●(and {body1 body2 ……})

bodyを順に評価し，評価値がnilとなった時点で実行を打ち切ってnilを返します．bodyが尽きるまでnilとならなければ，最後のbodyの評価値を返します．bodyがひとつも存在しなければtを返します．

●(or {body1 body2 ……})

bodyを順に評価し，nilでない評価値が得られた時点で実行を打ち切ってその値を返します．bodyが尽きるまでnilしか得られなければ，nilを返します．bodyがひとつも存在しなければnilを返します．

●(let varlist {body1 body2 ……})

局所変数リストvarlistにしたがってシンボルアトムのbindingをおこなった後，bodyを順に評価し，最後の値を返します．局所変数のbindingは，*prog*や*psetq*の場合のように並行しておこなわれます．

●(let* varlist {body1 body2 ……})

局所変数リストvarlistにしたがってシンボルアトムのbindingをおこなった後，bodyを順に評価し，最後の値を返します．局所変数のbindingは，*setq*の場合のように順におこなわれます．

●(catch 'tag-atm {body1 body2 ……})

bodyが順に評価されます．bodyの内部で，このcatchの持つtag-atmと等しいtag-atmを持ったthrowの評価が起こると，その段階でbodyの評価を終了し，〝throw〟された値をcatchの値として返します．throwはbodyの内部にあらわに見えている必要はありません．throwが起きなければ，catchは最後に評価した値を返します．

●(throw 'tag-atm {exp1 exp2 ……})

expを順に評価し，その最後の評価値を，このthrowと同じtag-atmを持つcatchに向けて返します．

●(catcherror {body1 body2 ……})

bodyが順に評価され，途中エラーが起きなければnilを返します．エラーが起きた場合は，そのエラーによる大域脱出をこの関数が捕まえます．したがって，bodyで起こったエラーはトップレベルまで戻りません．その場合，この関数は捕まえたエラーのエラーコードを返します．

● (eval 'exp ['env])

expを評価した値を返します.envが省略された場合には現在の環境で評価がおこなわれますが,これが与えられた場合にはenvを環境リストとして評価がおこなわれます.

```
% (eval '(plus 1 2 3))
6
```

● (apply 'func 'argument ['env])

argumentを関数funcの引数として与え，その評価値を返します．envが省略された場合には現在の環境で評価がおこなわれますが，これが与えられた場合にはenvを環境リストとして評価がおこなわれます.

```
% (apply 'plus '(1 2 3))
6
```

● (funcall 'func {arg1 arg2 ……})

引数arg1，arg2，……に関数funcを作用させて得られる値を返します.

```
% (funcall 'plus 1 2 3)
6
```

● (mapcar 'func arg-lis)

1引数関数funcをリストarg-lisの各要素に順に適用し,その結果を*list*でまとめたものを返します.

● (mapcan 'func arg-lis)

1引数関数funcをリストarg-lisの各要素に順に適用し，その結果を*nconc*でつないだものを返します.

● (mapcon 'func arg-lis)

1引数関数funcをリストarg-lisを先頭から順にcdrをとったものに適用し,その結果を*nconc*でつないだものを返します.

*mapcon*は，関数の適用がすべておこなわれてから*nconc*することが保証されています．また，*nconc*の際には各リストの端末を調べてから，前から順にポインタの書き換えをおこないます．したがって,循環リストが作成されてしまうような場合にも,*mapcon*の評価の途中で無限ループにおちいることはありません.

```
% (mapcon 'print '(a b c))
(a b c)
(b c)
(c)
(a b c c c c c ……      cの出力がずっと続く
```

4.2.9 リスト処理関数

● (car 'lis)

リスト lis の car 部分を返します．

● (cdr 'lis)

リスト lis の cdr 部分を返します．

● (cons 'exp1 'exp2)

exp1 を car に，exp2 を cdr に持つセルを作って返します．

● (append 'lis1 {'lis2 ……})

引数のリストをつなぎあわせたリストを返します．最後の引数を除くすべての引数は，一番上のレベルだけコピーされます．

```
% (setq a '((2 a) Q))
((2 a) Q)

% (setq b '(xyz (3 r)))
(xyz (3 r))

% (setq c (append a b))  ……この時(cdr (cdr c))と b は eq である．(car a)と(car c)も eq，(car
((2 a) Q xyz (3 r))        (cdr a))と(car (cdr c))も eq である
```

● (nconc 'lis1 {'lis2 ……})

引数のリストをつなぎあわせたリストを返します．最後の引数を除くすべての引数は，最後のセルの cdr の内容が書き換えられます．

```
% (setq a '((2 a) Q))
((2 a) Q)
```

```
% (setq b '(xyz (3 r)))
(xyz (3 r))

% (setq c (nconc a b))   ……この時bはもとのままであるが，aの値は((2 a) Q xyz (3 r))に書
((2 a) Q xyz (3 r))        き換えられる．aとcはeqである
```

● (list {'exp1 'exp2……})

exp1, exp2, ……を要素とするリストを返します．

● (reverse 'lis)

リストlisの要素の順番を逆順にしたリストを作成して返します．

● (assoc 'key-atm 'lis)

リストlisは次のような構造のリスト(連想リスト)でなくてはなりません．

```
((key-atm1 . value-exp1) (key-atm2 . value-exp2) ……)
```

assocはリストを順に見ていき，指定されたkey-atmと等しいkey-atmが見つかれば，そのセルを返します．

```
% (assoc 'marvel '((show . 23) (marvel . 20) (shot . 25)))
(marvel . 20)
```

● (rplaca 'lis 'exp)

第1引数であるリスト(ただしnilは許されない)のcarをexpへのポインタに書き換えます．

● (rplacd 'lis 'exp)

第1引数であるリスト(ただしnilは許されない)のcdrをexpへのポインタに書き換えます．

● (length 'lis)

与えられたリストの要素数を返します．最後のセルのcdr要素は無視されます．nilの*length*は0となります．

4.2.10 述語関数

● (eq 'exp1 'exp2)

exp1とexp2が同じ構造体へのポインタを持つならばt，そうでなければnilを返します．したがって数値の比較には使うことはできません．また，表記上は同じに見えるリストも場合によっては等しくないことがあります．

● (equal 'exp1 'exp2 ……)

exp1，exp2，……がすべて同じS式を持つならばt，そうでなければnilを返します．数値の同一比較にはこの関数を使います．

```
% (setq zdo '(pin tan jun) tac (cons (car zdo) (cdr zdo)))
(pin tan jun)

% (eq zdo tac)   ……zdoとtacの値は表示上は同じ(pin tan jun)だが，そのセルが違うためeqでは
nil                ない

% (equal zdo tac) ……だがequalではある
t
```

● (atom 'exp)

expがアトムならtを，そうでなければnilを返します．argが数値やnilの時もtを返します．

● (null 'exp)

argがnilならtを，そうでなければnilを返します．

● (member 'exp 'lis)

expがリストlisの要素にあれば，lisのexp以降の部分を返します．lisの中にexpが見つからなければnilを返します．

● (numberp 'exp)

expが数値ならばt，そうでなければnilを返します．

● (zerop 'num)

数値numが整数の0，あるいは実数の0.0ならt，そうでなければnilを返します．

● (greaterp 'num1 'num2 ……)

num1＞num2＞……＞numnが成り立てばt，そうでなければnilを返します．整数と実数が混在してもかまいません．

● (lessp 'num1 'num2 ……)

num1＜num2＜……＜numnが成り立てばt，そうでなければnilを返します．整数と実数が混在してもかまいません．

4.2.11 数値関数

● (add1 'num)

num＋1の値を，numが整数型なら整数型で，numが実数型なら実数型で返します．

● (sub1 'num)

num－1の値を，numが整数型なら整数型で，numが実数型なら実数型で返します．

● (minus 'num)

－numの値を，numが整数型なら整数型で，numが実数型なら実数型で返します．

● (plus 'num1 {'num2 ……})

num1＋num2＋ …… ＋numnの値を返します．引数の中にひとつでも実数があれば，返す値は実数型になります．

● (times 'num1 {'num2 ……})

num1×num2× …… ×numnの値を返します．引数の中にひとつでも実数があれば，返す値は実数型になります．

● (difference 'num1 {'num2 ……})

num1－num2－ …… －numnの値を返します．引数の中にひとつでも実数があれば，返す値は実数型になります．

● (quotient 'num1 {'num2 ……})

num1/num2/ …… /numnの値を返します．引数の中にひとつでも実数があれば，返す値は実数型になります．

● (remainder 'fix-num1　{'fix-num2 ……})

fix-num1 mod fix-num2 mod …… mod fix-numn の値を返します．引数はすべて整数型でなくてはなりません．

● (divide 'fix-num1　{'fix-num2 ……})

除算を行い，その商と余りをリストにしたもの，すなわち(list (quotient fix1 fix2 ……) (remainder fix1 fix2 ……))を返します．

4.2.12 文字操作関数

● (ascii 'exp)

exp がシンボルアトムなら，その印字名の先頭文字の文字コードを，exp が数値なら，その数値に当たる文字コードの文字を印字名とするシンボルアトムを返します．arg は1バイトしかチェックされないので，2バイトの漢字コードを与えても，漢字に変換されることはありません．

● (implode 'atm-lis)

atm-lis はシンボルアトムを要素とするリストでなくてはなりません．atm-lis の各要素の印字名をつなぎ合わせたシンボルアトムを返します．

```
% (implode '(zndoko tam jun))
zndokotamjun
```

● (explode 'atm)

シンボルアトム atm の印字名を1文字ごとに分解し，それらをリストにして返します．漢字は1文字として展開されます．

```
% (explode 'nue は魔物)
(n u e は 魔 物)
```

● (alen 'atm)

文字アトムの印字名の文字列の長さをバイト単位で返します．漢字などの全角文字は2つに数えます．

```
% (alen 'nue は nue)
8
```

● (intern 'atm1 {'atm2 ……})

oblist に登録されていないシンボルアトム atm1, atm2, ……を oblist に登録します. すでに同じ印字名をもつシンボルアトムが oblist に存在する時はエラーとなります. oblist に登録されたシンボルアトム atm1 を返します.

● (remob 'atm1 {'atm2 ……})

oblist に登録されているシンボルアトム atm1, atm2, ……を oblist から抹消します. atm1, atm2, ……が oblist に存在しない時はエラーとなります. oblist から抹消されたシンボルアトム atm1 を返します.

● (gensym)

評価されるごとに, oblist に登録されていないシンボルアトムを発生します. その印字名は

g000, g001, ……

というように, 〝g〟の後ろに3桁の10進数字を並べたものになり, g999 の次は g000 に戻ります.

4.2.13 入出力関数

入出力関数のほとんどは, ファイルディスクリプタを引数に含めることができます. ファイルディスクリプタは0以上の整数で, オープンされたファイルに付けられた固有の番号です. ファイルディスクリプタを入出力関数に渡すことで入出力をそのファイルからおこなうことができます. この時, 0, 1, 2 番のファイルディスクリプタは初めから次のものに確定しており, 新しくオープンしたファイルには3番以降のファイルディスクリプタが与えられます.

0 : 標準入力(コンソール)
1 : 標準出力(コンソール)
2 : 標準エラー出力

● (load 'filename-atm1 {'filename-atm2 ……})

指定したすべてのファイルからS式を読み込み評価します(返す値は t). ファイル名の拡張子を省略すると 〝.lsp〟 が補われます.

● (print 'exp ['fd-num])

exp を改行付きで出力し, exp を評価値として返します. ファイルディスクリプタ fd-num が指定されれば, そのファイルに対して出力し, 省略されれば現在の出力先に対しておこないます.

exp内のシンボルアトムの印字名に特殊文字が含まれる場合,およびシンボルアトムの印字名が数字から始まる場合には，その前にエスケープ文字〝¥〟がつけられます.

コントロール文字は〝#¥007〟のように出力されます.

oblistに登録されていないシンボルアトムは〝#：zndoko〟のように出力されます.

● (prin1 'exp ['fd-num])

expを出力し，expを評価値として返しますが，改行はおこないません．それ以外は*print*と同様です.

● (princ 'exp ['fd-num])

expを出力し，expを評価値として返します．改行はおこないません．シンボルアトムの印字名はそのまま出力されます.

```
% (setq beep (ascii 7))
#¥007

% (prog nil (princ beep) (princ beep) (princ beep))
nil  ……3回ベルが鳴る
```

● (terpri ['fd-num])

改行のみをおこなう.

● (read ['fd-num])

ファイルディスクリプタfd-numが指定されればそのファイルから，fdが省略されれば現在の入力先からS式を読み込み，そのS式を返します.

● (readch ['fd-num])

1文字入力を行い，読み込んだ文字を印字名に持つシンボルアトムを返します.

● (format 'fd 'ctrl-atm {'exp1 'exp2 ……})

書式付き出力を行います．fdはファイルディスクリプタで，出力はfdによって示されるファイルへ送られます．fdがnilの場合は，現在の出力先に対して出力がおこなわれます.

ctrl-atmは出力の書式を指定するもので，この印字名の中に，〝~〟(チルダ)で始まり，後に示すいくつかの〝変換文字〟で終わる変換指定を含めることで，そのうしろの引数exp1，exp2，……を，その変換指定の部分に指定した形で展開することができます.

＜変換文字＞

a　対応する引数を princ 出力する．引数は任意の S 式．

s　対応する引数を prin1 出力する．引数は任意の S 式．

c　引数は整数でなくてはならない．引数に相当する文字コードをもつ文字が展開される．

d　引数は整数でなくてはならない．引数を 10 進数で出力する．

o　引数は整数でなくてはならない．引数を 8 進数で出力する．

x　引数は整数でなくてはならない．引数を 16 進数で出力する．

e　引数は浮動小数点型でなくてはならない．引数を ［-］ n.nnnnnnE ［-］ mm の形に変換する．デフォルトでは小数点以下は 6 桁．

f　引数は浮動小数点型でなくてはならない．［-］nnnn.nnnnnn の形に変換する．小数点以下はデフォルトでは 6 桁．

g　引数は浮動小数点型でなくてはならない．e，f オプションのうち，短くなるほうに変換する．小数点以下はデフォルトで 6 桁．

p　引数は数値でなくてはならない．引数が整数型の 1 に等しければ，何もここには展開されない．そうでなければ，〝s〟（小文字の S）が出力される．

@p　引数は数値でなくてはならない．引数が整数型の 1 に等しければ，〝y〟（小文字の Y）が出力される．そうでなければ，〝ies〟が出力される．

n　改行コード（CR＋LF）が出力される．

t　タブが出力される．

r　CR のみが出力される．

b　バックスペースが出力される．

＜フィールド指定＞

a オプションでシンボルアトムを印字する場合，および，d,o,x,e,f,g オプションで数値を出力する場合はその出力にフィールド指定をすることができる．その方法は以下のとおり．

・チルダと変換文字の間に 10 進数 n をはさむことで，n 桁分確保される．
・n の前に〝ー〟（マイナス記号）を置くと左詰めになる．
・n の前に〝0〟を置くと，余分な空白の代わりに〝0〟が展開される．
・浮動小数点型の出力の場合は n のうしろに〝，m〟（m は 10 進数）を付けて小数点以下の桁数指定をすることができる．

フィールド指定を必要としない変換文字に指定をしても，それは無効となる．また，精度指定に用いる〝,〟は特殊文字なのでエスケープ文字〝¥〟か〝|〟を使う必要がある（ctrl-atm 全体を〝|〟で囲って置くことが望ましい）．

<継続オプション>

変換文字の直前に〝:〟を置くことにより，そのひとつ前の変換に使った引数を再度使用することができる．第1引数以前にこのオプションが使われた時は，nil が引数にとられる．

例：

```
% (format nil #:x~10ax 'zdogn)
x       zdognx

% (format nil #:|~c~:c~:c| 7)
nil         ……3回ビープ音がする

% (de nue (n) (format nil #:|I have ~d lil~:@p.~n| n))
nue

% (nue 1)
I have 1 lily.
nil

% (nue 3)
I have 3 lilies.
nil
```

● (open 'filename-atm ['mode-atm])

指定されたモードでファイルをオープンします．ファイル名は file の印字名で指定し，オープンに成功すればファイルディスクリプタを返します．失敗すればエラーとなります．mode-atm を省略すると自動的にリードテキストモードになります．

mode-atm		モード
r	:	リードテキスト(デフォルト)
w	:	ライトテキスト
a	:	アペンドテキスト
rb	:	リードバイナリ
wb	:	ライトバイナリ
ab	:	アペンドバイナリ

● (close 'fd-num)

ファイルディスクリプタ fd-num で示されるファイルをクローズし，引数の fd-num と同じ数値を返します．特にライトモード，アペンドモードでは *close* をしない限り，ファイルの更新は保証されないので，注意が必要です．0 から 2 の fd はシステムに予約されたものであるためクローズすることはできません．

現在の入力(あるいは出力)が向いているファイルを閉じると，入力(出力)は自動的にコンソールに戻ります．

● (fmode 'fd-num)

ファイルディスクリプタ fd-num で示されるファイルのモードを，*open* の時の mode と同じシンボルアトムの形で返します．

```
% (fmode 2)
w
```

● (curin)

現在の入力がどのファイルに向いているか，ファイルディスクリプタで返します．たとえば，コンソールならば 0 になります．

● (curout)

現在の出力がどのファイルに向いているか，ファイルディスクリプタで返します．たとえば，コンソールならば 1 になります．

● (dirin 'fd-num)

入力先をファイルディスクリプタ fd-num で示されるファイルにリダイレクトします．返す値は fd-num です．

● (dirout 'fd-num)

出力先をファイルディスクリプタ fd-num で示されるファイルにリダイレクトします．返す値は fd-num です．

4.2.14 その他の関数

● (oblist)

oblist のコピーを作って返します.

● (prompt 'atm)

プロンプトをシンボルアトム atm の印字名に切り替えます. ただし, atm が t の時は標準プロンプト〝% 〟になります.

● (reclaim)

ガベージコレクタを強制起動します.

● (verbos 'exp)

ガベージコレクタの出すメッセージの on/off をおこないます. exp が nil 以外ならメッセージ出力を表示し, exp が nil の時はメッセージを出さないようにします.

● (trace funcname-atm1 {funcname-atm2 ……})

引数として与えられた関数がトレースされるよう印をつけます. 評価値として, 与えられた関数名のリストを返します. トレースされるよう印のつけられた関数は, これ以降評価されるたびに, 渡された引数と返す評価値を次のような形式で出力します.

```
% (de fact (n) (cond ((zerop n) 1) (t (times n (fact (sub1 n))))))
fact

% (trace fact)
(fact)

% (fact 2)
-> fact : args 2
--> fact : args 1
---> fact : args 0
<--- fact : value 1
<-- fact : value 1
<- fact : value 2
2
```

● (untrace funcname-atm1 {funcname-atm2 ……})

引数として与えられた関数が以降トレースされないように印をはずします.

● (exec 'proc {'arg1 'arg2 ……})

子プロセスを実行します.子プロセスに引数を渡す時はプロセス名procの後にそれらを続けます.

● (quit)

Will o'Lispから抜けて,OSに戻ります.その際,まだオープンされていたファイルがあればすべてクローズします.

4.3 機能の拡張とその方針

本書でLispを作成する目的は,処理系を作るという作業をとおして,その構造をよりよく理解していただくことです.しかし,Will o'Lispは比較的小型の処理系ではあるのですが,それでも前項で見ていただいたように,かなり広範な機能を含んでいます.

この全ソースリストをいきなり提示されても,なかなか全体を見通すことはできないでしょう.そこで本書では,Lispインタープリタとして最小限の機能を持つWill o'Lispの〝核〟の部分をまず作成することにします.そして,その基本的システムを徐々に機能拡張することによって,最終的に完全なシステムへと成長させていきます.

この機能の拡張は,次に示す7段階の手順を踏んでおこなわれます.

①基本システムの作成(UNDEADバージョン)

ここでは,Will o'Lispの基本データ構造ならびに,リーダ,エバリュエータ,プリンタの基本的な機能を実現します.

②ガベージコレクタの追加(MADIバージョン)

このバージョンにおいて,Lisp処理系に不可欠なガベージコレクタの機能が組み込まれます.

③ prog関数の追加(MALORバージョン)

prog関数が組み込まれ,それに伴って,catch関数とthrow関数による大域脱出機能,let関数によるローカル変数などが使用可能となります.

④ファイル I/O 機能の追加(CALFO バージョン)

ファイル I/O 関係の関数が組み込まれ, OS 上のエディタで作成したプログラムのロードや, データのファイルへの保存が可能となります.

⑤ Lisp の常備関数の作成(MONTINO バージョン)

ここまでのバージョンでは, インタープリタの構造を説明することがおもな目的であり, Lisp が当然備えているべき関数も. ほとんど揃っていませんでした. そこで, いくつかの関数を追加し, Lisp としてのの機能 UP をはかります.

⑥マップ関数と汎関数の作成(MADALTO バージョン)

関数引数機能, 汎関数の機能が追加されます.

⑦マクロ機能の追加(MAX バージョン)

マクロの追加と, いくつかの補助機能の拡張がおこなわれます.

5章 Lispの基本構造を作成する

本章からいよいよ実際に処理系のコーディングにかかります．しかし，前章で設計した仕様を完全に満たす処理系をいきなり作成しても，理解するのは量的に困難であると思われます．そこで本章では，まず Will o'Lisp の核となる最小限のシステムを作成し，それを通して Lisp 処理系の構造を理解していただこうと思います．

5.1 データ構造の定義 ―lisp.h― (List 5.1)

lisp.hは，すべてのモジュールにインクルードされる重要なヘッダファイルです．この中でWill o'Lispの扱うデータ構造，定数，大域変数の宣言がおこなわれます．

5.1.1 漢字を扱うためのデータ型の宣言 (11行〜12行)

Will o'Lispの仕様では，シンボルアトム名にシフトJIS漢字を含むことを許しています．しかしシフトJISコードは漢字やカタカナに0x80以上の文字コードを用いているため，コンパイラによっては，符号つきの〝char〟型を用いて文字を操作すると文字の比較の時などに不都合を生ずるかもしれません．

そこで，本書のプログラムでは〝unsigned char〟型を〝uchar〟と型定義し，このuchar型を用いて文字を扱うことにします．また，ucharへのポインタ型〝uchar *〟を〝STR〟と型定義し，特に文字列へのポインタを表す時に，このSTRを使います．

したがって，本書に掲載したプログラムにおいて，char型はかならずしも文字を扱うものではありません．それらは単に〝小さな整数値〟を扱うためのデータ型としてとらえてください．

5.1.2 Will o'Lispのデータ構造 (14行〜42行)

Will o'Lispのデータタイプは，セル，シンボルアトム，数値アトムの3種類です．これらはFig. 5.1のような構造体として，それぞれ宣言されています．

```
セル構造体
struct cell
  {id; car; cdr}
   id: char型数値   car, cdr: セルへのポインタ

数値アトム構造体
struct num
  {id; value}
   id: char型数値   value: 数値(共用体)

シンボルアトム構造体
struct atom
  {id; value; plist; name; ftype; fptr}
   id: char型数値
   value, plist: セルへのポインタ
   name: 文字列へのポインタ
   ftype: char型数値
   fptr: セルへのポインタ
```

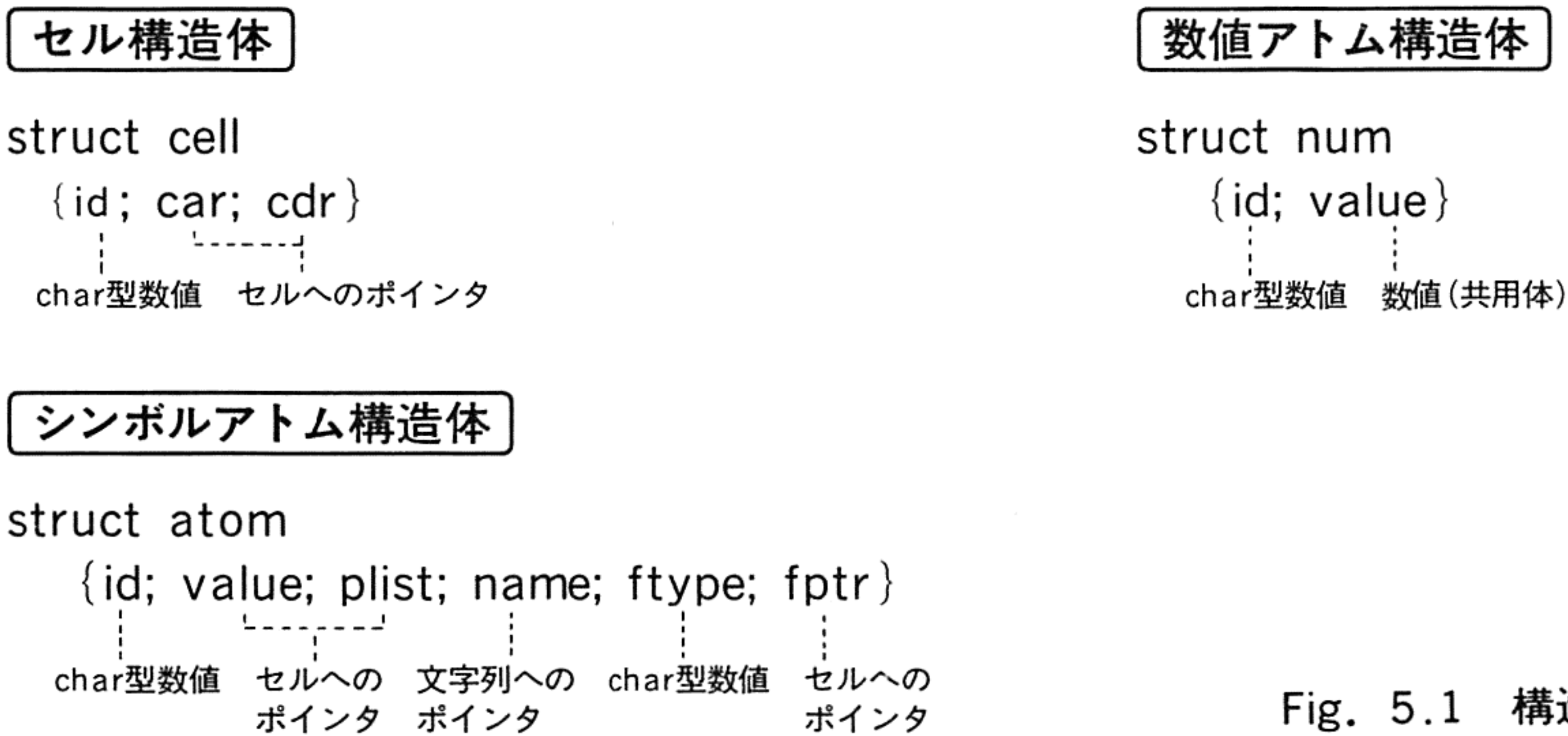



Fig. 5.1 構造体とそのid

どの構造体も，その構造体の種類を示す〝id〟情報をメンバに持っています．どの場合もidは最初のメンバで，しかも同じchar型ですから，それが置かれている位置(オフセット)はどの構造体でも同一です．したがって，

ptr->id

と記述することで，ptrがどの構造体を指していてもidの値を取り出すことができ，その構造体が何であるのかを知ることができます．

本書のプログラムでは一般に，ある関数が下位の関数にLispのデータ構造を引数として渡す時，その引数が下位関数の必要とする条件を満たしているかどうかのチェックをおこないません．そのため，リストを必要とする関数にシンボルアトムを授けてしまうこともあり得ます．このような場合に備え，引数を受け取る側は，それが自分の必要とするデータ型へのポインタであるかどうかを判断しなければなりません．

ところが，ポインタというのは基本的には単なるアドレスですから，それが何へのポインタであるのか直接にはわかりません．Cのプログラム上でのポインタ宣言は，あくまでもそのデータをどのように〝使う〟かということを言っているにすぎないのです．その中身が宣言どおりになっているとは限りません．実際，本書のプログラムでは，セルへのポインタと宣言された変数がシンボルアトムへのポインタを隠し持っていることは少なくありません．

そこで，〝宣言という外観〟に惑わされずに〝本音〟を見抜く方法が必要になるのです．Will o' Lispでは各々の構造体にidという身分証明書を持たせ，この問題を解決しているわけです．

このid方式には，タイプチェックがメンバ参照だけでよい，データ型の追加が容易にできる，などのメリットがあるのですが，その代わりメモリ効率が犠牲になります．idはchar型で宣言してありますから，各構造体はidを用意しない方法より少なくとも1バイト余計に大きくなりますし，さらにコンパイラによっては，メモリアクセスの関係で構造体内のchar型変数がint型で展開されますので，その計算機での1ワード長になることもあります．この場合，セル構造体の3分の1に相当するメモリをid用に食われてしまう事態も起こり得ます．

そこでidを使う以外の方法を考えてみると，まず思いつくのはポインタの表すアドレスの大きさを比較し，それがどの領域に属するかを決定する方法です(Fig. 5.2)．しかし，この方法では，そのポインタの表す対象が何であるかを決めるために専用の関数を用意しなくてはなりません．その関数は，アドレスがA以上かつBより小さかったらセルで，B以上C未満だったらアトムで，というようにif文の列になるでしょう．しかもその関数の呼び出される回数は，たいへんな数になることが予想されます．繁雑な関数呼び出しのせいで実行速度に不安が残るので，この方法は採用できません．

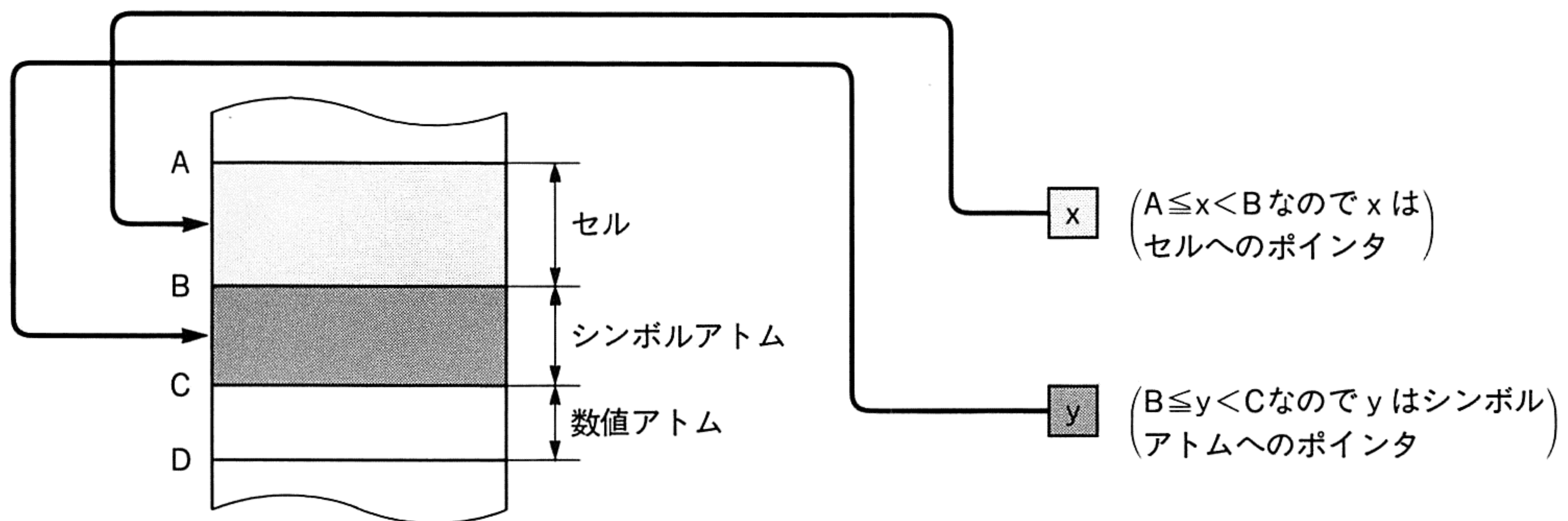



Fig. 5.2　アドレスの大きさによるポインタの比較

次にこういう方法が考えられます．セルの大きさは，4バイト，8バイトなどの偶数バイトですから，ポインタの下位の2～3ビットは実際には使われずに空いています．そこで，この部分にidにあたる情報を詰め込んでしまうのです．一見よさそうな案ですが，この方法ではポインタによる参照をおこなうたびに，いちいちマスクをかける必要があります．アセンブラならともかく，これはCの仕事ではありません．

では，いっそのことポインタを使うのを止めて，配列とインデックスだけで記述したらどうでしょう．

```
cell->cdr->car
```

と書く代わりに，

```
car [ cdr [ cell ] ]
```

と書くのです．メモリ効率もよく面白い方法ではありますし，BASICでもFORTRANでも書ける，その意味では一般的な方法ともいえます．しかしながら，本書の目的のひとつは，Cの持つポインタとキャストによりロマンとファンタジーに満ちあふれたひとつの世界を創造することですから，このような方法はあえてとらないことにしましょう．

●セルの構造

Will o'LispのセルはFig. 5.3に示す構造を持ちます．先頭のメンバは前述のようにidであり，続く2つのメンバcarとcdrは構造体cellを指すポインタです．このように定義の中で定義されている構造自身を持ち出すことを不安に思う人がいるかもしれませんが，2つのメンバcarとcdrはポインタ型ですから，そのサイズは確定しており，〝struct cell *〟はポインタの指しているもののタイプを添えているにすぎず，問題はありません．

以後，〝セルをつなぐ〟，〝アトムをつなぐ〟など，〝つなぐ〟という言葉が使われますが，それは，構造体のポインタ型のメンバに，次にくる構造体へのポインタ(すなわちその構造体のアドレス)を代入する作業を意味します．

id ←―― _CELLにセット

carへのポインタ

cdrへのポインタ

```
struct cell {
    char   id;
    struct cell  *car;
    struct cell  *cdr;
```

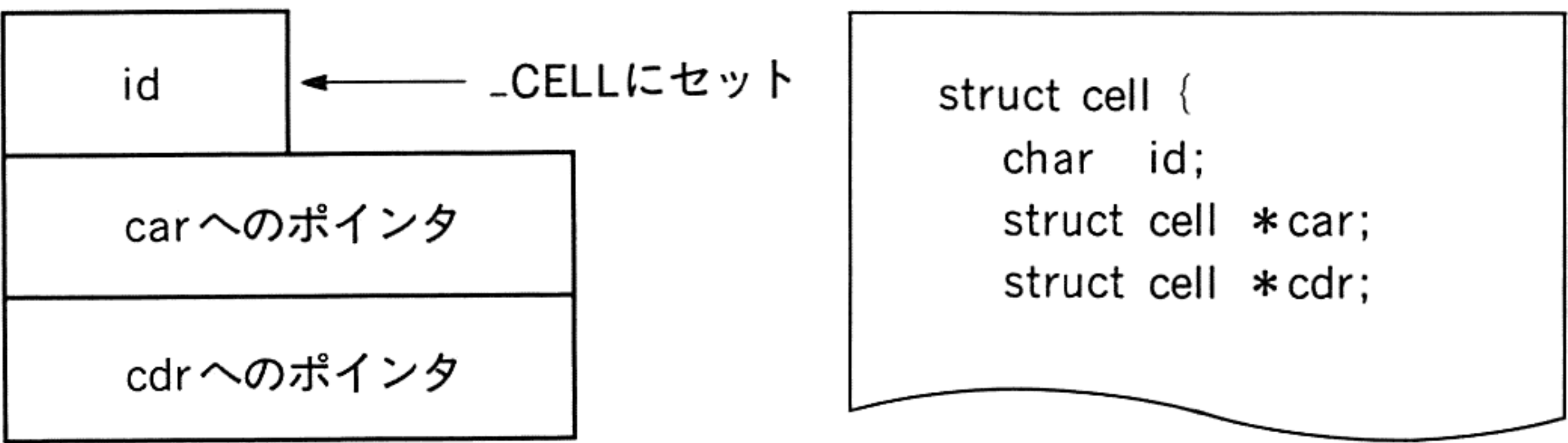



Fig. 5.3 セル構造体

さて，Lisp処理系が使用できるセルの数は，その処理系の性能を表す目安のひとつです．Will o'Lispはどのくらいの数のセルを扱えるでしょうか．ここでちょっと見積ってみましょう．

8086上のCコンパイラでは，一般に4バイトのfarポインタを使うラージモデルと，2バイトのnearポインタを使うスモールモデルを使い分けられます．ラージモデルを使い，さらにidは2バイトに展開されたとすると，セルの大きさは，

id(2バイト)+car(4バイト)+cdr(4バイト)　　ラージモデルのセルサイズ

の計10バイトになります．1セグメント64Kバイトをセル領域に開放したとすると，65536/10＝6553個のセルが使用できます．スモールモデルを使うと，セルの大きさは

id(2バイト)+car(2バイト)+cdr(2バイト)　　スモールモデルのセルサイズ

と合計6バイトで，先ほどよりは小回りがききそうな気もしますが，この場合は同じセグメント内にアトムやら数値やら文字やらの領域をとらなくてはなりませんので，セルの数はかえって減ってしまい，4000セルくらいが限度となります．

●シンボルアトムの構造

セルの構造体に比べ，シンボルアトムの構造体はたくさんのメンバを持っています．LISP1.5などではアトムは特別な構造を持たず，セルのcarにあたる部分にアトムであることを示す印を付け，cdrの部分に印字名や値や属性値といったいろいろな情報を属性リストとして持っていました(Fig. 5.4)．しかし，最近のLispではこのような方法でアトムを実現することは，あまりないようです．Will o'Lispでもアクセス効率を考えて，それらの情報をシンボルアトム構造体の中に取り込んでいます(Fig. 5.5)．

◎：アトムであることを示すマーク

PNAME　印字名へ　EXPR　関数ボディへ

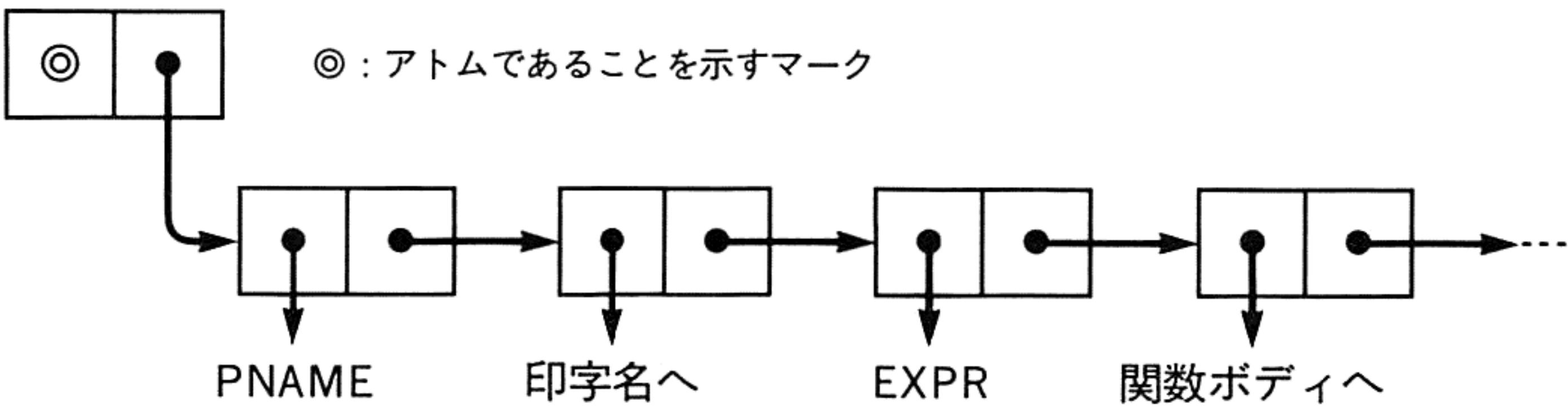



Fig. 5.4 Lisp1.5のシンボルアトムの構造

id ← _ATOMにセット
シンボルの値へのポインタ
属性リストへのポインタ
印字名の文字列へのポインタ
関数タイプ
関数本体へのポインタ

```
struct atom {
    char   id;
    struct cell  *value;
    struct cell  *plist;
    char    *name;
    char   ftype;
    struct cell  *func;
};
```

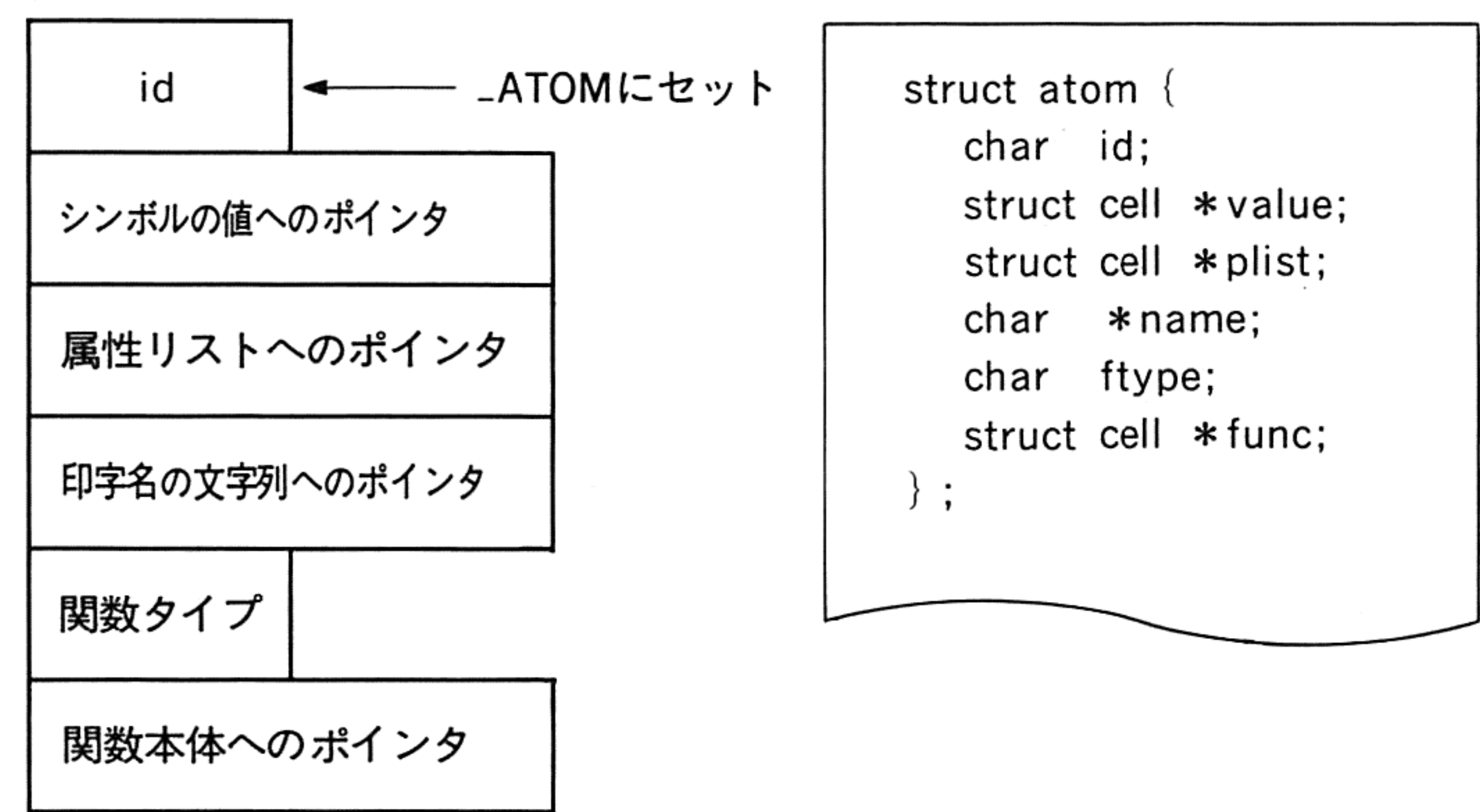



Fig. 5.5 Will o'Lisp のシンボルアトムの構造

初めのメンバは前述のとおり，構造体がシンボルアトムであることを示す〝id〞で，次にそのシンボルアトムの持つ値へのポインタ〝value〞が位置します．このポインタはセル，シンボルアトム，数値アトムのいずれかを指します．

Lisp 処理系によっては，使用者が何らかの値をシンボルアトムに与えない限り，そのシンボルアトムからの値の取り出しはエラーとなりますが，Will o'Lisp ではオートクォートと呼ばれる働きにより，任意のシンボルアトムはあらかじめ自分自身を値として与えられています．したがって，多くの場合この value というメンバは，自分が含まれるシンボルアトム構造体へのポインタを格納していることでしょう．

次のメンバは属性リストへのポインタ〝plist〞です．属性リストは，アトムに与えられた属性を，**Fig. 5.6** のように奇数番目に属性名，偶数番目に属性値を要素とする交代リストの形で記憶しています．シンボルアトムが何も属性値を持たない時は plist は nil を指しています．

アトム
属性 1　属性 2
plist
属性名 1　属性値 1　属性名 2　属性値 2

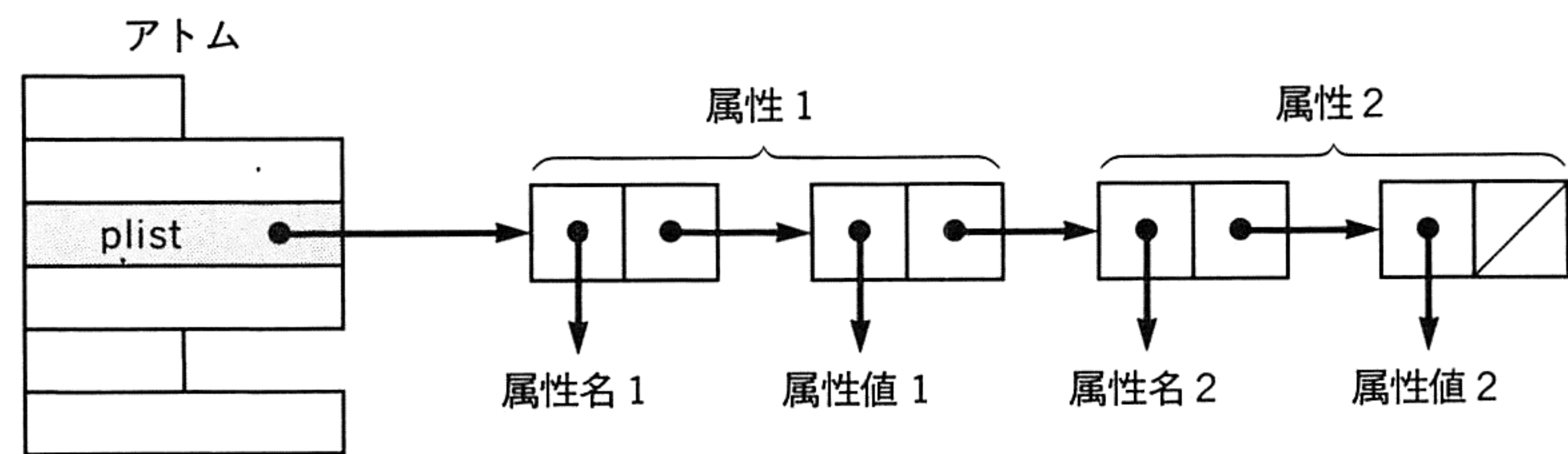



Fig. 5.6 属性リスト

そして印字名(シンボルアトム名の文字列)へのポインタ〝name〞が続きます．これはそのシンボルアトムの印字名が記憶されている文字領域を指しています．

シンボルアトムは関数名としても使われます．シンボルアトム構造体の最後の 2 つのメンバは，その関数のために存在します．〝ftype〞というメンバが，expr，fexpr，subr，fsubr といった関数のタイプを表し，〝fptr〞がその関数本体へのポインタとなっています．

●数値アトムの構造

Will o'Lisp が扱える数値には整数型と実数型の 2 種類がありますが，構造体の種類としては 1 種類だけです．それは，Fig. 5.7 に示すように，共用体を使い整数型と実数型を切り替えて使っているからです．整数型の領域と実数型の領域とを分けて確保する処理系もありますが，Will o' lisp ではメモリ効率を考えてこの方法を採ることにします．整数型は C の long にあたる 4 バイト長，実数型は double にあたる 8 バイト長です．さらに，この共用体の中には数値アトムの構造体を鎖状につなぐためのポインタ〝ptr〟の宣言も含まれています(「5.4 自由領域の管理」参照)．

id ← _FIX / _FLT のいずれか

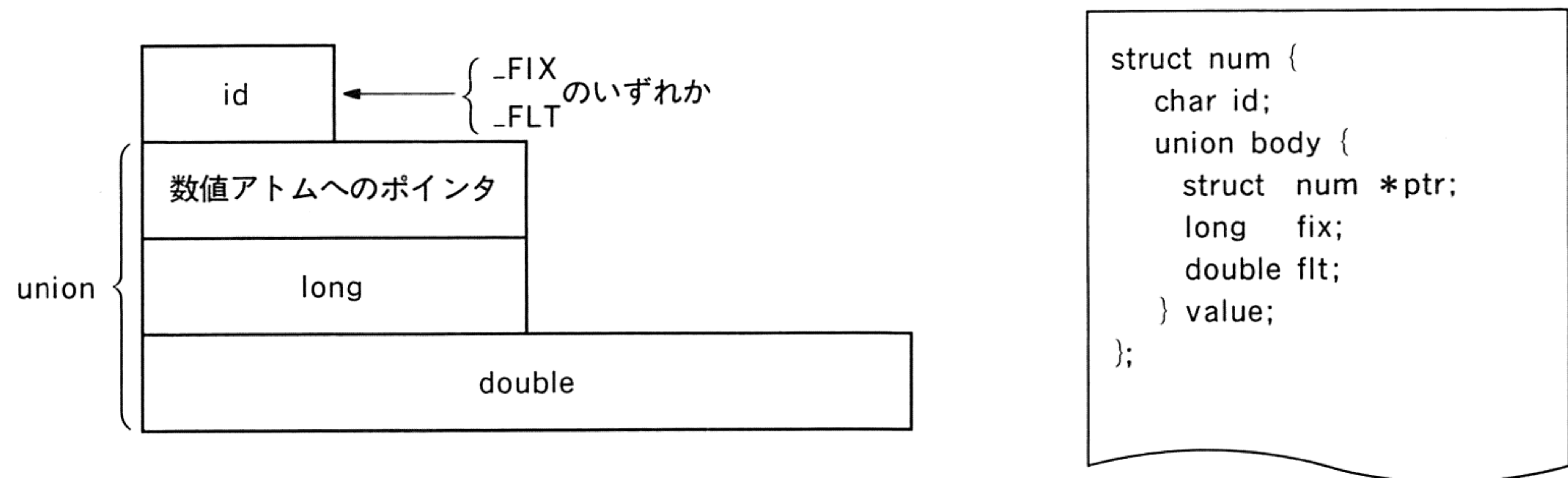
union: 数値アトムへのポインタ / long / double

```
struct num {
   char id;
   union body {
     struct  num *ptr;
     long    fix;
     double flt;
   } value;
};
```

Fig. 5.7 数値アトムの構造

構造体をどの型で用いるかは先頭にあるメンバ id によって区別し，union 内で long にとるか double にとるかを切り替えます．たとえば，数値アトムへのポインタ np から整数型として値を取り出す時は，

```
np->value.fix
```

のようにすればよいのです．

●ポインタ型の型定義

以後説明するプログラム中で，関数の返す値や変数の多くは〝構造体へのポインタ〟を値とします．そこで，ポインタ型というものを先に型定義しておくと後々便利です．セルを例にとると，ここではセルの構造体 CELL へのポインタを表す型として，CELLP を宣言してあります．

セルへのポインタを値として持つ変数 cp を宣言するならば，次の①のようになります．

```
CELLP cp;            ……①
```

これは，もちろん以下の②あるいは③と同じ意味です．

```
CELL *cp;            ……②
struct cell *cp;     ……③
```

本書のプログラム中では①の CELLP による宣言の形を使うことにします．またセルと同様に，シンボルアトムや数値アトムへのポインタを示す型名も〝ATOMP〞，〝NUMP〞というように宣言してあります．

5.1.3 定数とマクロの定義 (44行〜130行)

データの識別に使われるタグには，構造体の判別に使われる id(各構造体の先頭メンバ)と関数タイプの区別のための ftype(シンボルアトム構造体のメンバ)の 2 種類があります．

●構造体 id

〝_CELL〞に始まる 4 つの定数は，構造体判別用の id に対して与えられます．これらは次のような意味を持っています．

_CELL	この id を持つ構造体はセルである
_ATOM	この id を持つ構造体はシンボルアトムである
_FIX	この id を持つ構造体は整数アトムである
_FLT	この id を持つ構造体は実数アトムである

●関数タイプ

〝_NFUNC〞からの 4 つの定数は，シンボルアトムの ftype に与えられ，関数タイプの区別に使われます．これらはビットごとに次のような意味があります．

最下位ビット	そのシンボルアトムが関数として定義されていない時に ON
第 1 ビット	その関数が C で記述されている時に ON
第 2 ビット	引数が評価されるタイプの関数の場合に ON

たとえば〝_SUBR〞の値は 0x06 で，第 1 ビットと第 2 ビットが ON ですから，これは〝C で記述されていて引数が評価される関数〞を示すことになります．

その次の 3 つの定数は，上記のビットが ON であるかどうかを調べるためのマスクです．このマスクと ftype のビットごとの and をとった結果が 0 となるかどうかで，目的の関数がどんな性質を持っているかがわかります．

●領域の大きさ

その次に領域の確保に関する定義がいくつか並んでいます．そのうち，特に初めの 4 つは，この Lisp 処理系が使うメモリ量に大きく関係します．もし，メモリが足りなくて Lisp が起動でき

なかったら，この4つの定義をもう少し小さいものに書き換えてコンパイルし直さなくてはなりません．

CELLSIZ	セル領域の大きさ
ATOMSIZ	シンボルアトム領域の大きさ
STRSIZ	文字列領域の大きさ
NUMSIZ	数値アトム領域の大きさ

なお，これらの数はその領域の総バイト数ではなく，その構造体を〝いくつ〟確保するかの個数です．たとえばセルの場合は，

sizeof(CELL) * CELLSIZ

がセル領域の実際の総バイト数になります．

その他の定数の意味は以下のとおりです．これらの定義は自分が必要とするシステムの特性に応じて自由に書き換えてください．

TABLESIZ	oblist 用のハッシュテーブルの数
LINESIZ	1回に取り込める入力行のバイト数
NAMLEN	シンボルアトムの印字名の最大長

●フラグ用の定数

領域確保に関する定義の次は各種のフラグの定義です．

ESCON	シンボルアトムを表示する際に，特殊文字のエスケープをおこなうよう指定する
ESCOFF	特殊文字をエスケープせず，そのまま出力するよう指定する
ON	文字列の読み込み中に，いまマルチプルエスケープをおこなっているかどうかを示す
OFF	マルチプルエスケープ中ではないことを示す
TOP	最も外側のリストを読み込んでいることを示す(スーパーカッコの処理に用いる)
UNDER	リストの要素であるリストを読み込んでいることを示す(　　〃　　)
TRUE	論理値の〝真〟を示す
FALSE	論理値の〝偽〟を示す
NONERR	エラーが発生していないことを示す
ERR	エラーが発生したことを示す
ERROK	エラーが発生したが，その表示はもう終わったことを示す

●エラーチェック用マクロの定義

〝ec〞のマクロ定義は，エラー発生時の大域脱出に使われるものです．このマクロの使用法に関しては，「5.6 エラー処理」の項を参照してください．

●エラー番号の設定

次に並ぶ define 文は，エラー番号を意味のある名前と結びつけています．たとえば，〝Cell area used up(セル領域を使い尽くした)〞というエラーを発生させる場合，本来は

```
error(4);
```

を実行するのですが，ここでおこなった定数定義によって，同じことを

```
error(CELLUP);
```

という，はるかにわかりやすい方法で記述できます．このエラー番号の並び方は，error.c ファイル中のエラーメッセージ配列に対応しています．

5.1.4　大域変数宣言　(132 行〜141 行)

最後に大域変数の宣言に関するファイルを取り込むインクルード文があります．もしこの lisp.h がインクルードされる前に # define 文で〝MAIN〞が宣言されていれば，大域変数の実体がある〝defvar.h〞が，そうでなければ大域変数の参照宣言がある〝var.h〞が読み込まれます．この 2 つのインクルードファイルには，次の内容が含まれています．

● defvar.h　(List 5.2)

ここには複数のファイルで使われるグローバルな変数の宣言があります．

それらはファイルポインタ，システムアトム，oblist，各領域の先頭ポインタ，入力バッファと文字ポインタ，エラーフラグ用の変数です．

この defvar.h ファイルは，main.c モジュールでのみインクルードされます．

● var.h　(List 5.3)

defvar.h の各変数の先頭に〝extern〞を付けて，外部参照の宣言に変えただけです．main.c 以外のモジュールでは，この var.h がインクルードされます．

5.2 S式の読み込み －read.c－ (List 5.4)

ここでは，Lispの最も基本となる〝S式の読み込み〟をおこなうリーダ関数を作成します．Lispのリーダは，入力ファイルから文字列を受け取り，S式としての構造を解析してメモリの中に格納する機能を備えていなければなりません．このための関数をまとめたファイルがread.cです．

5.2.1 文字列の読み込み

● getstr() (13行～19行)

これは文字列を読み込むための最も下位レベルの入力関数です．現在の入力先がコンソールならばプロンプトを画面に出力するのも，この関数の仕事です．

Will o'Lispでは，大域変数promptが指しているアトムの印字名をプロンプトとすることにしています(promptの値は，main.cファイルのmk_sys_atoms()関数の中で設定される)．UNDEADバージョンではプロンプトはそのまま固定されますが，CALFO以降のバージョンでは，この変数が指すアトムを変更することにより，任意のプロンプト出力が可能です．

文字列の入力は，fgets()を用いて入力バッファoneline[]に文字列を読み込み，文字ポインタtxtp(txtpは大域変数として定義されている)をoneline[]の頭に持ってくればでき上がりです．通常はそのtxtpの値，すなわち文字バッファの先頭へのポインタが返されますが，EOFを読み込んだ場合には，fgets()の定義にしたがってNULLが返されます．

なお，ここで使われているcur_fpiというファイルポインタは現在の入力先を示すもので，UNDEADバージョンにおいてはstdinと同義です．直接stdinから文字列を読み込まず，このようなファイルポインタをとおして入力をおこなっているのは，実は後のバージョンでおこなう入出力機能拡張のための布石です．

● skipspace() (21行～31行)

文字バッファの先頭にある不用な空白文字を飛ばし，次の文字までポインタtxtpを進める関数です．その時txtpの指す文字が通常の文字を指していればTRUEを返しますが，txtpが行末(ヌル文字〝¥0〟か，コメントの始まりを示すセミコロン〝;〟)に達していたら，次の文字列を読み込んで，空白文字が先頭にあるかどうかを再びチェックします．ただし，1行を読み込む関数getstr()がNULLを返した時はEOFを表しているので，skipspace()もNULLを返して上位の関数にそれを伝えます．

5.2.2 S式の判別と振り分け

読み込んだ文字列を解析してS式の構造を作り出し，メモリの中に格納する作業は，次の3つの機能があれば実現できます．

①シンボルアトム(文字アトム)を作る．
②数値アトムを作る．
③リストを作る．

Lispにおいて，入力された文字列の中からこれらの3つの要素(シンボルアトム，数値，リスト)を識別するのは，非常に簡単です．というのは，リストはカッコから始まり，数値は数字から始まり，それ以外はアトムになるといった具合に，LispのS式は自分が何であるかをその先頭の文字によって強烈に主張するからです．つまり，受け取った文字列の先頭の文字が，その文字列に対しておこなうべき処理を決定してくれるのです．したがって，Lispのリーダは，入力バッファ内の先頭文字を見てその要素が何であるのかを調べ，新しい要素を作り出せばよいのです．

この①～③に当たる関数は，作った対象へのポインタを返すことにします．アトムを作る関数はアトムへのポインタ，リストを作る関数はそのリストの先頭のセルへのポインタといった具合です．しかし，C言語の関数は，ある決まったタイプの値しか返すことができません．そこで，リーダはかならずセルへのポインタを返すものとして宣言しておきます．Will o'Lispでは，各オブジェクトにid用のタグが付いていますから，このようにしてもリーダを呼んだ側で〝何への〟ポインタを受け取ったのか判断できますし，必要なら適当なキャストをかければ問題はありません．

● read_s()　(37行～50行)

上記の割り振りをおこない，生成したオブジェクトへのポインタを返す関数です．この関数がLispのリーダの本体であり，read_s()を呼ぶごとに，ひとつのS式が読み込まれます．

引数であるlevelは，スーパーカッコの処理のために使われるフラグです．スーパーカッコは実際にはリスト生成関数が処理するので，read_s()はただlevelの値を下位の関数に渡しているにすぎません．

最初のskipspace()は次に読むべき文字列の頭出しをおこないます．この時，もしNULLが返ってくればEOFを読んだということですから，それ以上の読み込みを中止してeofread(〝EOF〟を表すアトムへのポインタ)を返します．

文字列が無事に読めたら場合分けに移ります．まず，num()で数値かどうかを調べます．数値かどうかは，1文字目に符号がある場合を考えて，文字列の1文字目と2文字目を見なければなりません．そこで，num()には文字列を指すポインタtxtpを引数として渡し，2文字目を見られる

ようにします．もし数値であれば，数値を生成する関数mk_num()を呼びますが，これは数値へのポインタを返す関数なので，(CELLP)のキャストをかけてread_s()の戻り値とします．

次に特殊文字かどうかをisesc()で調べ，特殊文字ならばescopt()を呼びます．特殊文字とは，普通はシンボルアトムの名前に含むことのできない文字を指し，Will o'Lispには13種類存在します．リストの始まりを示すカッコも特殊文字ですから，リストはここで捕まります．

さて数値が抜け，リストが抜けましたから，残るはシンボルアトムです．ここでは，isprkana()*1で英数字・記号・カナかどうかをチェックし，iskanji()*2でシフトJIS漢字コードの上位バイトに該当するかどうかを判断します．このどちらかにあてはまれば，シンボルアトムへのポインタを返す関数ret_atom()を呼びます．この関数はアトムへのポインタを返してくるので，やはり(CELLP)のキャストをかけてread_s()の戻り値とします．

ここまでの条件を満たさない文字はリーダの受け付けることのできない文字ですので，エラーとなります．

● escopt()　(52行～65行)

S式が特殊文字で始まる場合は，さらにescopt()という関数で処理の振り分けをおこないます．このUNDEADバージョンでS式の先頭にくることが許される特殊文字は，〝(″(左カッコ)，〝[″(左角カッコ)，〝|″(バーティカルバー)，〝¥″(円記号またはバックスラッシュ)の4つです．

もし特殊文字がカッコならば，リストを作る関数mk_list()を呼び出し，〝¥″や〝|″はシンボルアトムの印字名に特殊文字を含むためのエスケープ文字ですので，ret_atom()を呼びます．

それ以外の特殊文字には，〝'″(クォート)，〝#″(シャープサイン)，〝^″(バッククォート：本来は〝‘″を使用するべきだが，PC-9801にはこの記号が用意されていないため，〝^″で代用する)などがあります．これらは以降のバージョンにおいてマクロ文字としての働きを持つことになりますが，UNDEADバージョンではまだ何の機能も持たず，読み込むとエラーになります．

5.2.3 数値アトムの読み込み

Will o'Lispの扱う数値にはfloat型とfix型の2種類があります．この両者はまったく異なるものとして認識され，たとえばfix型の〝1″とfloat型の〝1.0″はequalにはなりません．したがって数値アトムを読み込む際にも，float型あるいはfix型をきちんと区別して数値を作り出す必要があります．一応，手順を示しますと，

---

*1 *2　MS-C日本語バージョンではctype.hの中でマクロとして定義されているため，これらの関数を作成する必要はない．

(1) 符号のチェック
(2) 整数部の取り出し
(3) 小数点以下の取り出し
(4) 指数部の取り出し
(5) 数値構造体に値をセット

となります．(3)や(4)の処理を通過したものはfloat型，そして小数点がなく，指数を表す〝e″や〝E″の文字もなくて(2)の処理から(5)へ飛んだものはfix型になります．

● mk_num() (67行〜107行)

この関数は，上記の手順にしたがって数値の型チェックをおこないながら文字バッファからnumbuf[]という作業領域に数字列をコピーし，最後にその型に合わせてatol()またはatof()を用いて実際の数値を得ます．

newnum()は自由数値アトム領域からひとつの数値アトム構造体を取り出し，そのポインタを返す関数です(newnum()はgbc.cファイルに含まれている)．ただし，newnum()が返したアトムのidは_FIX(整数型)にセットされているので，それに実数を代入する時にはidの変更が必要です．newnum()の後ろの〝ec；″は数値アトムを使い尽くしたエラーのための脱出用です(「5.6 エラー処理」参照)．

5.2.4 シンボルアトムの読み込み

リーダが読み込んだシンボルアトムは，かならずoblistというリストに登録されます．oblistは過去に現れたシンボルアトムのカタログのようなものです．すでに存在しているシンボルアトムと同じ印字名のシンボルアトムが入力文中に現れた時には，Lispはもはやアトムを作る作業はせず，単にoblistを参照するだけですませます．したがって同じ名前を持つシンボルアトムはひとつしか存在していません．

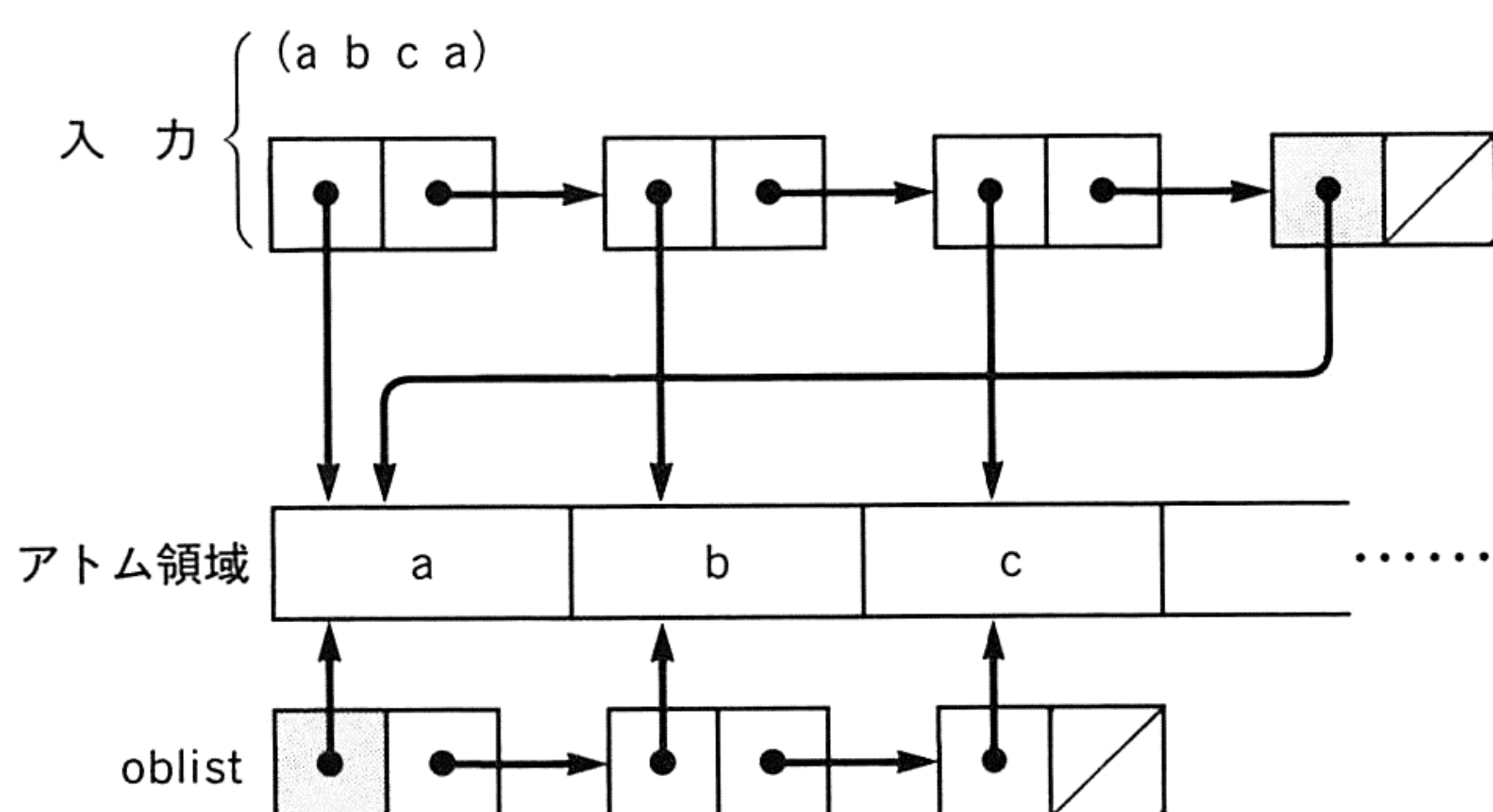



Fig. 5.8 シンボルアトムはひとつ

リーダがシンボルアトムにでくわしたら，まず oblist の検索にいき，同じ印字名を持つものがあれば，そのシンボルアトムへのポインタを返します．見つからなければそこで初めてシンボルアトムを作る作業をするのです(Fig. 5.8)．これは数値アトムの場合と違います．数値アトムの時は過去に同じ値を持ったものが存在していたかどうかを調べることなく，毎回新しいアトムを作る作業を行います．

● ret_atom()　(109行～123行)

シンボルアトムへのポインタを取り出す時の最上位に位置する関数です．read_s()から直接呼び出されます．

まず初めに，シンボルアトムの印字名にあたる部分を getname()によって入力バッファから取り出し，文字配列 nambuf [] に格納します．nambuf [] のサイズは，lisp.h の中で定義された NAMLEN(シンボルアトムの印字名の最大長)にヌル文字の分を加えただけ確保してあります．

次に，oblist を検索する関数 old_atom()を呼び，oblist の中に同じ印字名を持つシンボルアトムが見つかれば，そのポインタを返します．見つからなければアトムを作り出して oblist へ登録する関数 mk_atom()を呼び，そこで新しく作られたシンボルアトムへのポインタを返します．

ところで，この小さな Lisp 上でも，プログラムを動かしていると，登録されているシンボルアトムの数は，もともとある関数名やラベルの数も含めて数百にもなることがあります．これをひとつの oblist の中にすべて直線的に並べておくと，検索をする上で効率が悪いことはいうまでもありません．そこで，ハッシュ法を用いて oblist を分割します．いくつに分割するかは，lisp.h の中で定義された TABLESIZ で決定されます．

ただし，このくらいの規模の処理系では，あまり神経質にシンボルアトムのテーブルへのちらばり方を気にする必要もないので，ハッシュ関数はごく簡単なものでかまいません．具体的には文字列の各文字のコードの和をとり，それと TABLESIZ との剰余をキーと決めることにしましょう．

同じキーを持ったシンボルアトムができた時，それらはひとつのリストにまとめられて管理されます．新しいシンボルアトムが追加される時は，その古いテーブルのリストに，新しいシンボルアトムをぶらさげたセルを継ぎ足せばよいのです．

これらの個々のリストは，C のプログラム上では oblist [TABLESIZ] というポインタの配列が押さえています．n 番目のテーブルのリストの最初のセルを oblist[n]が指し，n 番目のテーブルに何もシンボルアトムが存在しない時は，それは nil を指しています(Fig. 5.9)．

oblist[0] → SYM1 → SYM2 → SYM3
oblist[1] → SYM4
oblist[2] → nil
oblist[TABLESIZ-I] → SYMY → SYMZ

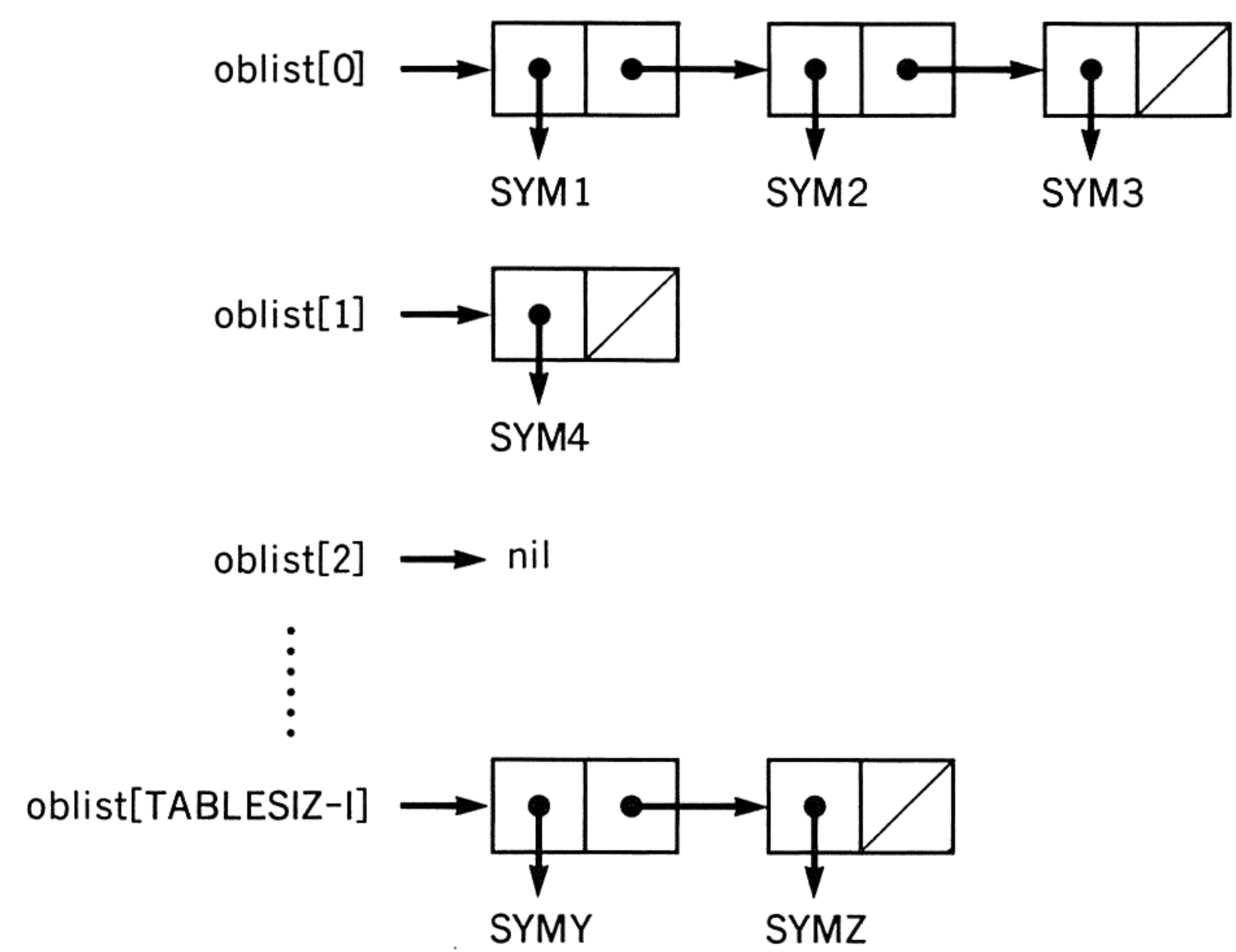



Fig. 5.9 oblist のテーブル構造

● hash() (125行〜132行)

文字列を与えると，そのハッシュ値を返す関数です．上記のように，各文字コードの総和をとり，TABLESIZ で割った余りを計算しています．

● old_atom() (134行〜147行)

文字列を引数に取り，その文字列と同じ印字名を持つシンボルアトムが oblist にあるかどうか調べます．引数の文字列からハッシュ用のキーを計算し，そのキーによって与えられる oblist テーブルのリストについて，nil が出てくるまで cdr をたぐりながら，セルの car にあるアトムの印字名と引数の文字列を比較していきます．途中，同じものがあったらそのシンボルアトムへのポインタを返し，nil にぶつかるまで見つからなかったら，シンボルアトムがなかったことを表す NULL を返します．

● mk_atom() (149行〜157行)

まず mk_sub() を呼び出して新しいシンボルアトムをひとつ作ります．そこでエラーが起こらなければ，作ったアトムへのポインタを oblist に登録します．実際の登録作業は，intern() を呼び出しておこないます．

● mk_sub() (159行〜179行)

引数として与えられた文字列を印字名に持つようなシンボルアトムを作り出す関数です．まず，未使用の文字領域が，引数として受け取った文字列を格納できるだけ残っているかどうかを調べます．文字領域が使いつくされていれば〝String area used up〟のエラーになりますが，文字領

域に余裕があれば印字名の文字列をコピーします．

次に，newatom()を呼び，未使用シンボルアトム領域からシンボルアトムの構造体をひとつ取り出します(newatom()はgbc.cファイルに含まれている)．無事に取り出せたら，値，印字名，属性リスト，関数タイプを初期化します．

シンボルアトムの値には自分自身へのポインタを与えておきます．これによって，初めて登場したシンボルアトムもエラーになることなく評価することができます．この機能はオートクォートと呼ばれています．

● intern() （181行〜192行）

シンボルアトムへのポインタを引数に取り，それをoblistに登録します．まずnewcell()によって未使用のセルをひとつ取り出して，そのセルのcarにシンボルアトムをぶら下げます(Fig. 5.10)．次に，印字名からそのハッシュ値keyを求め，keyの番号にあたるoblistの先頭に先ほど作ったセルを追加します(Fig. 5.11)．

自由リストから取り出されたセル
新しいシンボルをぶら下げる
SYM
NEW SYM
…… アトム領域

Fig. 5.10 シンボルアトム登録の手順1

oblist[key]
nil
新しいシンボルアトム
シンボルアトム1
シンボルアトム2
oblist[key]
新しいシンボルアトム
シンボルアトム1
シンボルアトム2

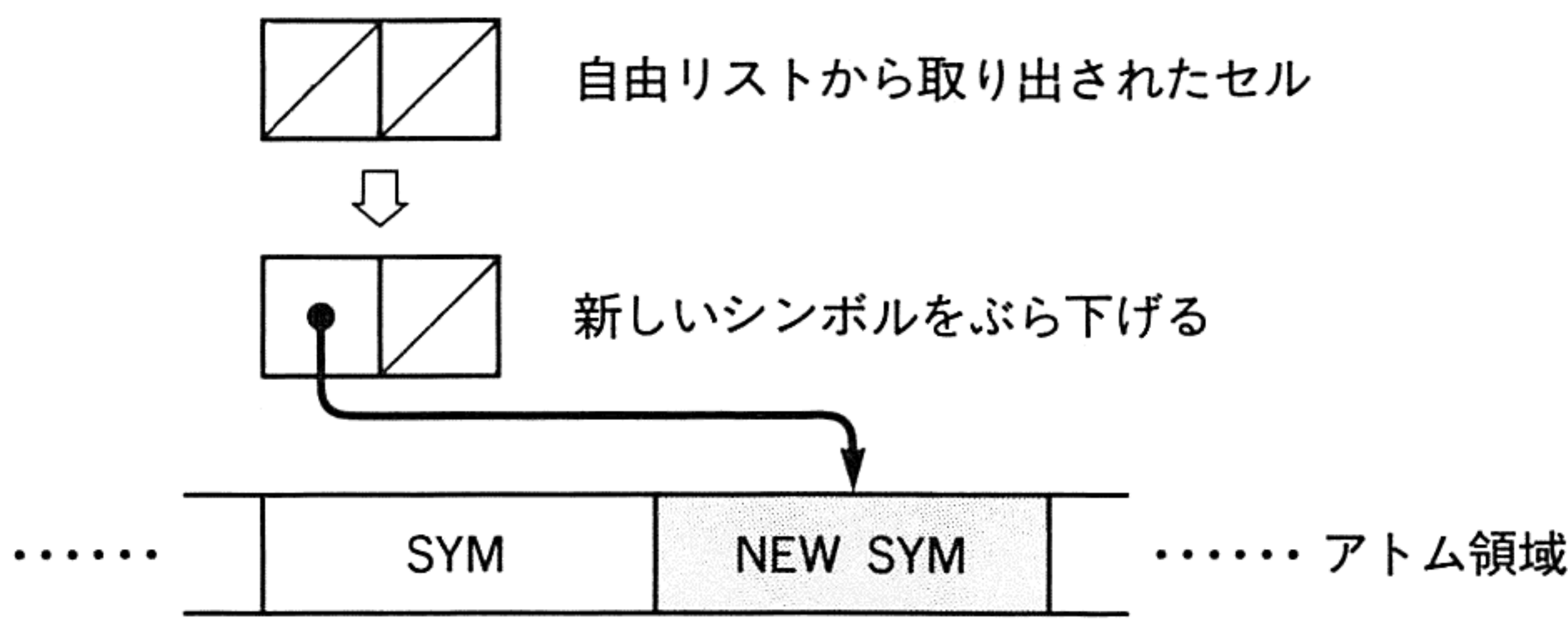
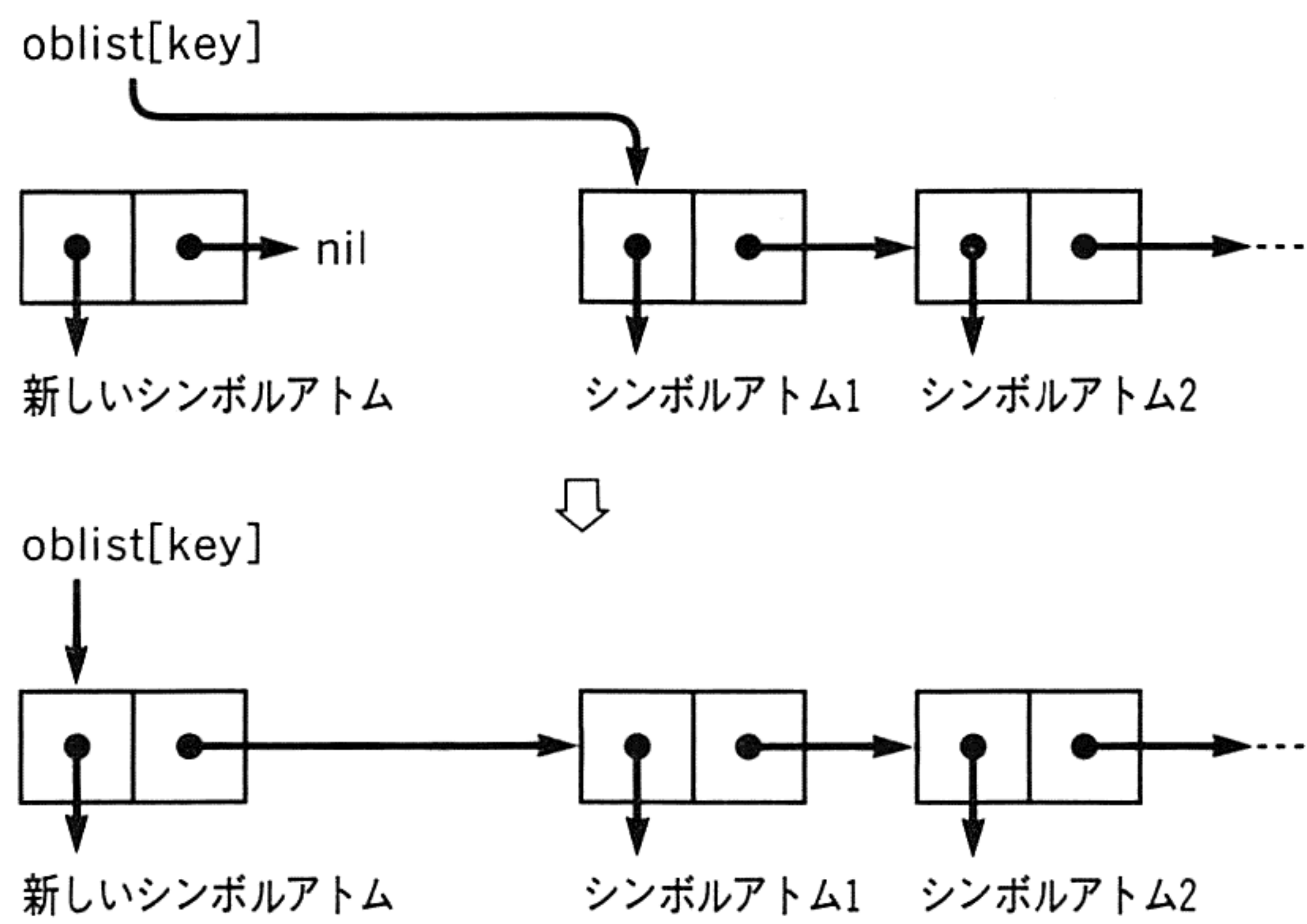



Fig. 5.11 シンボルアトム登録の手順2

5.2.5 名前の読み込み

Will o'Lispでは，エスケープ文字を用いて，シンボルアトム名に特殊文字を含ませることができます．そのため，入力バッファから名前を切り出すには少々面倒な処理が必要となります．

まず，特殊文字のエスケープ規則を以下に書きます．

① "¥" の後にある特殊文字，数字は普通の文字として扱われる(1文字エスケープ)．
② "|" と "|" の間に囲まれた特殊文字は普通の文字として扱われる(マルチプルエスケープ)．
③ 改行コード "¥n" がエスケープされた時は，シンボルアトム名が次の行へ継続することを表す．

例：

```
% ¥123
¥123                 ……"123"を印字名に持つ文字アトム．123という数値ではないことを示すため，出力時にも"¥"が付いている(「5.3 S式の表示」参照)

%|W E R D N A|
W¥ E¥ R¥ D¥ N¥ A    ……マルチプルエスケープ

% Restore-¥
% Strength           ……シンボルアトム名の継続
Restore-Strength
```

このうち②のマルチプルエスケープに関する規則は，Common Lispのそれに準拠しています．たとえば，MacLispでは

|Zndoko||tamtam|

は|Zndoko|と|tamtam|という2つのシンボルアトムとして読まれますが，Common Lispや，このWill o'Lispでは，|Zndokotamtam|というひとつのシンボルアトムとなります．つまり，空白文字以外の文字が続く限り，名前は連続しているとみなされるのです．ただし，偶数番目の"|"と奇数番目の"|"の間ではマルチプルエスケープの機能は働きません．下の例を見てください．

```
% |a()ity|
a¥(¥)ity                ……通常のマルチプルエスケープ

% |a|nil|ity|
anility                 ……"|"で区切られていても，文字が続いている限り名前は連続しているとみな
                          される

% |a|()|ity|
a                       ……だが，偶数番目の"|"と奇数番目の"|"の間では特殊文字のエスケー
                          プはおこなわれない
nil

ity
```

● getname()　(194 行～238 行)

さて，実際のプログラムに移ります．変数 ifesc はマルチプルエスケープ中かどうかのフラグです．まず，シンボルアトム名が許容された長さを超えないように全体を for 文の中に入れておきます．その中でおこなう文字列からの名前切り出し手順は以下のようになります．

(1)マルチプルエスケープ文字がくるごとにフラグの切り替えをする．"|"が連続することを考えて while 文になっている．

(2)マルチプルエスケープの状態でなければ(7)へ．

(3)1 文字エスケープ文字だったら入力バッファの文字ポインタを進めて(7)へ．

(4)シンボルアトム名の区切りとなる空白や特殊文字でなければ(7)へ．

(5)コメントを表す";"ならば，その行は終わりなので，リーダに次の行の入力をさせるために";"をヌル文字に置き換えて，入力バッファが空になったことを明示しておく．

(6)区切り文字が現れたので複写先の文字配列にヌル文字を移し，全過程の終わり．

(7)エスケープされた改行コードは無視する規則のため，行の終わりだったら次の行の入力にいく．入力を受け取ったら(1)へ(普通，行の終わりの"¥n"は(4)で検知され(6)で抜けてしまう．ここで"¥n"を検出するのは(3)からここへきた時のみ)．

(8)漢字コードがきたら 2 文字複写して(1)へ．

(9)英数字，記号，仮名文字でなかったら，エラーを返す．

(10) 1 文字複写して(1)へ．

5.2.6 リストを作る

リストを作るには，セルのcar部分にリストの要素をぶら下げながらcdrを次のセルにつないでいく作業を繰り返します．リストの要素は任意のS式ですから，その読み込みには前述のリーダ関数read_s()が利用できます．リストの中にリストがあるような多重にネストした構造でも，リーダを再帰的に呼び出し，〝子リスト〟を作らせて親リストのセルのcar部分にぶらさげればよいのです．

read_s()は，それがどのようなオブジェクトを読み込んだかにかかわらず，そのS式へのポインタを返すものとして定義しました．したがって，リストを作る時は，要素の区切りとなる空白文字を読み飛ばして，何かの要素にぶつかるごとにリーダを再帰させてポインタを取り出せばよいのです．リストを作る時にはそのリストの要素の種類が何であるのかを知る必要はなく，ポインタだけが問題となるのですから．それ以外に，このリストを作る関数に必要とされることは，対応する1組のカッコを閉じ，入力バッファの先頭文字ポインタをリストの終わりまできちんと進めておくことです．

たとえば，

(tam jun (pin zdo nue) wool)

を読み込む場合は，まずtamとjunを読み，junを2つ目のセルにつないだ時点で次の要素に〝(〟が現れたので，リスト作成関数を再帰的に呼び出し〝(pin zdo nue)〟のリストを作らせて，そのポインタを受取りセルにつなぎます．この時には，入力バッファの文字ポインタは〝nue)〟の次のスペースまで進んでいるので，次の〝wool〟を取り込み，最後に〝)〟に出会ったのでリストを閉じ，〝tam〟のつながっているこのリストの先頭のセルを返すのです．

普通はこのような処理をすればよいのですが，Will o'Lispにはスーパーカッコがサポートされていますので，その処理を考えなくてはなりません(ちなみにスーパーカッコはCommon Lispにはありません)．

スーパーカッコ〝]〟は最も近い〝[〟か，それがなければ最も外側の〝(〟までのリストをすべて閉じてしまうもので，リストの最後の〝))……))〟をひとつの〝]〟で代用することができます．うまく使えば入力の手間が省け，たいへん便利なものです．

```
% (car (quote (di badi madi]   ……(di badi madi)と(quote …… )と(car …… )の3つを〝]〟で閉じる
di

% (cdr [quote (di badi madi])……(di badi madi)と(quote …… )の2つを閉じる
(badi madi)
```

このスーパーカッコの処理も含めたリスト作成のアルゴリズムは次のようになります．

（1）リストの始まりが〝[″だったら，スーパーカッコで始まることを示すフラグを立てる．
（2）空白文字を読みとばす．
（3）〝)″ がきたら要素のない空リストなので先頭文字ポインタを進めて nil を返す．
（4）〝]″の時も nil を返す．ただし，スーパーカッコで始まることを示すフラグが立っているか，最も外側のリストを作っている時にだけ，このカッコから先の文字へ進むことができるので，その時は先頭文字ポインタを進める(この 〝]″ にとどまっていれば上のレベルのリストが 〝)″ の代わりに使うことができる)．
（5）car 部がないのに 〝.″ がきたらエラー．
（6）リストの先頭のセルを newcell()で自由リストから確保し，car 部を取り込み，このセルにぶらさげる．この関数の返す値はこのセルへのポインタだから，ポインタを保存しておく．
（7）リストの終わりのカッコがきたら，終了処理(15)へ．
（8）〝.″ でなかったら，次のセルへいくので，(13)へ．
（9）空白文字を読みとばした上でカッコがきたら cdr がないのでエラー．
(10) リストの最後のセルの cdr をセットする．
(11) 空白文字を読みとばした上でカッコがなかったらエラー．
(12) 終了処理(15)へ．
(13) 次のセルを newcell()で確保し，要素を取り込み car にセットした後で，そのセルを前のセルの cdr につなぐ．
(14) 空白文字を読みとばして(7)へ．
(15) 閉じるカッコが〝]″の時，スーパーカッコで始まることを示すフラグが立ってないか，または，最も外側の〝(″を閉じるのでなければ，入力文字バッファの先頭文字ポインタを進めずに作ったリストの先頭のセルへのポインタを返す．((4)と同じ理由．)
(16) 先頭文字ポインタを進めて，リストを返す．

説明の最後にドット対の読み込みに触れておきます．Will o'Lisp ではピリオド〝.″は普通，ドット対を作るための特殊文字として扱われますが，数値入力中に現れた時は小数点としての扱いの方が優先されます．したがって，car 部が数値であるようなドット対を作ろうとする場合はピリオドの前にスペースを入れて小数点でないことを明記し，数値の読み込みを終了させておく必要があります．もちろん，このような扱いを受けるのはひとつ目のピリオドだけで，2つ目からは普通に区切り文字としての作用を持ちます．次のような例を参考にしてください．

```
(A.)        →  エラー
(.B)        →  エラー
(1.)        →  (1.000000) エラーにはならない
(.1)        →  エラー
(A.B)       →  (A . B)    ドット対になる
(1.2)       →  (1.200000) "." は小数点として解釈される
(1 .2)      →  (1 . 2)    ドット対になる
(A..B)      →  エラー
(1..2)      →  (1.000000 . 2) ひとつ目は小数点，2つ目はドットとして解釈される
(1.0.2)     →  (1.000000 . 2) 上と同様
(1. 0.2)    →  (1.000000 0.200000) リストになる
```

● mk_list() (240行～290行)

前述のアルゴリズムをCで記述したものですので，特に説明は必要ないでしょう．強いていえば，セルポインタcpがリストの先頭のセルを押さえていること，newcell()が切り出してきたセルはcarもcdrもnilを指していること，car，cdrをセルにセットする関数getcar()，getcdr()には，それらが内部で呼ぶread_s()の引数のために，リストの中の要素の取り出しであることを示す〝UNDER〟を引数として与えることの3点ぐらいでしょう．

● getcar() (292行～300行)

引数にセルをとり，そのcar部分にread_s()で取り出した要素をぶらさげます．プログラム中で，直接に

```
cp->car = read_s(level);
```

としないでいるのは，エラーが起きた時の保護のためです．一般の関数はエラーが起きるとNULLを返しますので，そのNULLをセルへつないでしまうのを防がなくてはならないからです．

● getcdr() (302行～312行)

getcar()とは逆にcdrに要素をセットします．前述のアルゴリズムでいえば(10)と(11)にあたりますので，(11)に相当するエラーチェックが入っています．

5.2.7 文字の種類の判別

● **num()** （314 行～321 行）

与えられた文字列が数値としての条件を満たしているかどうかを判断します．1文字目が数字ならTRUE，1文字目が符号で，かつ，2文字目が数字の場合もTRUEを返し，それ以外はFALSEを返します．この関数は，print.cのファイルの中でも使います．

● **isesc()** （323 行～342 行）

与えられた文字が特殊文字ならTRUEを，そうでなければFALSEを返します．〝#〟はマクロ文字，〝^〟と〝,〟はバッククォート用に，UNDEAD以降のバージョンで用いられます．バッククォートに〝^〟を使っているのは，PC-9801のキーボードにバッククォートがないので，仕方なく代用しているのです．

● **isprkana()** （344 行～350 行）

与えられた文字が英数字，記号，カナ文字だったらTRUEを，そうでなかったらFALSEを返します．〝0xa1〟から〝0xdf〟がカタカナのコードに対応しています．なお，MS-C日本語バージョンを使用している場合には，このisprkana()および，次の2つの関数iskanji()，iskanji2()は，同じ機能を持つマクロがctype.hの中で定義されていますから，削除してください．

● **iskanji()** （352 行～358 行）

与えられた文字がシフトJIS漢字コードの上位バイトに相当していればTRUEを返します．

● **iskanji2()** （360 行～366 行）

与えられた文字がシフトJIS漢字コードの下位バイトに相当していればTRUEを返します．

5.3 S式の表示 －print.c－ (List 5.5)

S式をプリントする作業は，すでにメモリの中にでき上がっているポインタによる構造を追いかけて目に見える形に表示していくだけのことです．構文解析の必要がありませんから，S式を読み込む作業よりはるかに楽ですし，また，再帰処理を用いることで簡潔に記述することができます．print.cファイルには，この〝S式の表示〟をおこなうための関数をまとめました．

5.3.1 リストを含むＳ式の表示

Lisp のプリンタ関数は，基本的に以下の規則に従います．

① 数値はその値を，シンボルアトムはその印字名を，出力する．
② リストの先頭には左カッコ〝(〟を置く．
③ リストの区切りは空白〝 〟で表す．
④ リストの終わりはセルの cdr 部にアトムが現れることで判別する．
⑤ リストの終わりの cdr は〝 . cdr)〟の形に出力する．
⑥ ただし，最後の cdr が nil の時は単に〝)〟でよい．

ここで④の規則について少し説明を加えておきましょう．これはいい替えれば Fig. 5.12 (a)のような〝リストを cdr に持つドット対〟は絶対に表示されないということを表しています．Lisp のプリンタは，できる限りドット対を省略した表示をおこなう約束になっており，このような構造は Fig. 5.12 (b)の形式で出力されるからです．

入力されたＳ式

(zakkai . (pin zndoko)) ························(a)

⇩ リーダ

内部構造

zakkai pin zndoko

⇩ プリンタ

出力されるＳ式

(zakkai pin zndoko) ····························(b)

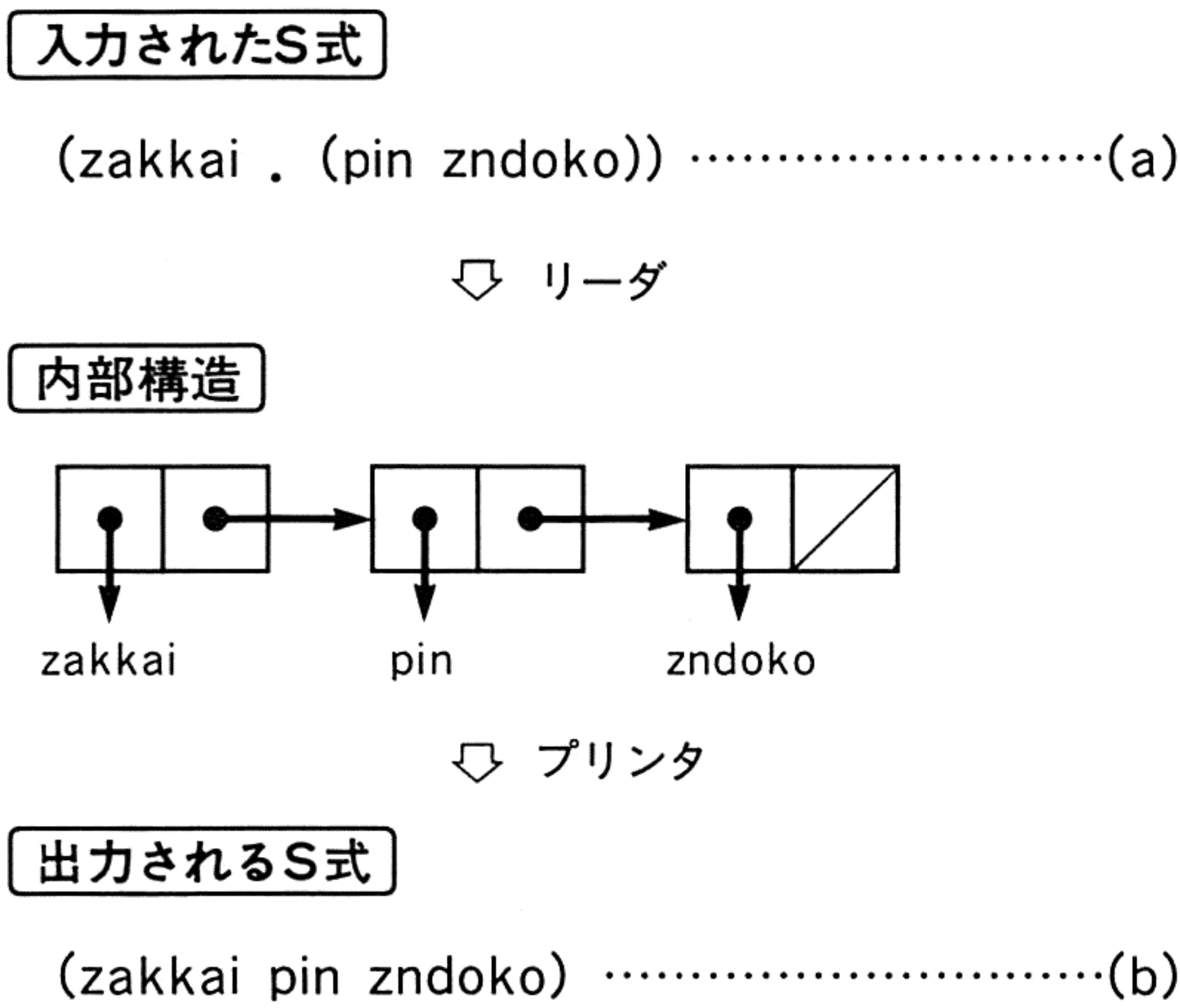



Fig. 5.12 プリンタはドット対の表示を避ける

● print_s() (9行～29行)

これは上記の規則のとおりにＳ式をプリントする関数であり，出力するＳ式へのポインタ cp と，出力形式を指定する mode の 2 つの引数を必要とします．この mode というのは，シンボルアトム名の表示を prinl 出力(Lisp リーダが読み込むのに都合のよい形式)とするのか，それとも princ 出力(人間が見るのに都合のよい形式)とするのかを指定するものです．両者の違いについては putstr ()のところでもういちど詳しく説明します．

さて，print_s()は，まず引数として渡されたポインタがセルを指しているかどうかを調べます．その結果がセルでなければポインタの先には1個のアトムが存在するだけですから，pri_atom()という関数にその表示をまかせて処理を終了します．

次に，cpの指しているものがセルの時は，まず左カッコ〝(〟を出力し，次にそのセルのcar部にあるS式(すなわちリストの要素)を表示しながらリストのcdrを次々とたどっていきます．要素を表示するにはprint_s()自身を再帰的に用いています．こうしてcdrをたどっていった結果，セルでないものが現れたら，それはリストの最後のcdrを示すアトムです．そこで，ドット対を表すピリオドとその〝最後のcdr〟を表示し，さらにカッコを閉じてリストの表示を終了します．ただしこの最後のcdrがnilであった場合には，単にカッコを閉じるだけでドット対の表示はおこないません．

なお，出力文の中にあるファイルポインタcur_fpoは，リーダの所で述べたcur_fpiと同様に，将来におけるFILE I/Oの拡張のための布石で，このUNDEADバージョンでは，stdoutとまったく同じ意味を持っています．

5.3.2　アトムの表示

Will o'Lispが扱うアトムには，整数，実数，シンボルアトムの3種類があります．Will o'Lispの仕様では，それらを次のような方法で表示することにしています．

整数：そのまま素直に表示する．

実数：2.0や3.0など，小数部分が0である〝区切りのよい値〟でも，整数と区別するために小数点〝.〟を省かない．

シンボルアトム：prin1出力形式と，princ出力形式の2種類の方法を用意する．prin1形式では，シンボルアトム名に特殊文字が含まれていた場合，エスケープ記号〝¥〟をその特殊文字の前に付けて出力するため，Lispリーダがその出力を再び正しく読み込むことができる．一方princ形式では，特殊文字を含む文字列でもエスケープ記号を付けずに出力するため，人間が読むのに適する表示がおこなえる．

● pri_atom()　(31行～44行)

3種のアトムは，Cのプログラム上ではすべて異なった実体を持ち，表示手段もそれぞれ異なっています．そのため，このpri_atom()では，まず引数として渡されてきたポインタがどの種類のアトムを指しているのかidタグを調べ，その結果によってswitch文で振り分けて個別に処理をおこないます．

整数アトムの場合はまったく簡単なもので，数値アトムの構造体からその値を取り出して10進出力するだけです．ただし，Will o'Lispの整数がlong型であることに注意し，

〝% ld〞

という倍精度表示の指定をしなければなりません．また，引数cpはセルへのポインタとして宣言されているので，それを数値として扱うには，

((NUMP)cp)->name

のようにキャストが必要です．

実数アトムの場合は，やはり数値アトム構造体から実数としての値を取り出し，fprintfの実数表示フォーマット，

〝%#. 6g〞

を用いて表示します．これは小数点以下6桁までの有効数字を表示し，またどんな場合でも小数点を省かないというものです．

そしてシンボルアトムならば，その印字名を次に説明するpurstr()関数へ送ります．なお，まずあり得ないことですが，何らかの原因でアトムのidが破壊された時，またはおかしなポインタが渡されてきた時のチェック用に，switchの最後にdefault文を付け，エラーを知らせるようにしておきました．

● putstr()　(46行～72行)

シンボルアトムの印字名を出力するための関数です．出力の仕方には2とおりあって，その指定は引数modeでおこないます．modeが定数ESCOFF(lisp.hで定義されている定数)に等しい時はprinc，それ以外はprin1のモードになります．

princは特殊文字とその他の文字を区別せずに表示する出力形式です．表示すべき文字列に対して何の判断もいりませんから，fprintfで印字名の文字列を出力するだけです．これに対し，prin1の場合は，特殊文字に対する配慮からいくつかの約束に注意しなければなりません．

① 数字で始まる印字名を持つシンボルアトムは，数値アトムと区別するために，先頭に〝¥〞を付ける．(例：　1234　→　¥1234)
② 特殊文字には，その直前に〝¥〞を付ける．(例：　Will o'Lisp　→　Will¥　o¥'Lisp)
③ コントロールコードなどの，英数字，記号，カタカナ，シフトJIS漢字コード以外の文字は，そのアスキーコードを〝#¥nnn〞の形で出力する．(例：　Ctrl-G　→　#¥007)

①の規則は num()を使って文字列の先頭だけでチェックします．num()は引数にとった文字列が〝数字列〟であれば TRUE を返しますから，fputc で〝¥〟を出力し，後は他の文字列と同様に扱います．

文字列を表示するアルゴリズムは，以下のようになります．②，③の規則と漢字処理を1文字毎にチェックしていくのです．

(1) 文字列の先頭がヌル文字〝¥0〟ならば，ヌル文字列を表す〝||〟を表示してすぐ終了．
(2) 次に出力するべき2文字がシフト JIS 漢字コードに相当しているかどうかを iskanji() および iskanji2()で調べ，該当していればその2バイトを fputc()で出力し，次のループに入る．
(3) (2)を突破したもので，コントロールコードなど，文字として表示できないものは，上記のように〝#¥nnn〟の形で10進表示し，次のループに入る．シフト JIS 漢字コードの上位バイトのみから成る1バイトの文字アトムが仮に存在してもここで捕まる．
(4) 特殊文字がきたら，〝¥〟を出力して，(5)に移る．
(5) 1文字出力する．
(6) 文字列の最後に達していなければ(2)へ戻り，次のループに入る．

5.4 自由領域の管理 －gbc.c－ (List 5.6)

Lisp が立上がった直後，初期化の作業によって，一定量のセル，文字・数値アトムの構造体のための領域が確保されます．これらはもちろん初めからすべてが参照を受けているわけではなく，そのほとんどが未使用の領域です．これを自由領域と呼ぶことにします．gbc.c ファイルには，その自由領域を管理するための関数が含まれています．それらの関数は，要求があれば自由領域から構造体をひとつ切り放し，それへのポインタを返します．たとえばリード関数がリストを読み込むために新しいセルが欲しいなら，この gbc.c ファイルにある〝セルを取り出す関数〟newcell()を呼べばよいのです．

現時点では，自由領域を使い尽くしてしまえば，もう永久に失われたままですが，MADI バージョン以降では，この gbc.c にガベージコレクタが追加され，失われた自由領域を回復させる重要な機能を持つことになります．

5.4.1 セルを取り出す

セル領域の未使用部分は〝自由リスト〟と呼ばれるひとつのリストの形につながれて保持されており，このリストの先頭のセルは大域変数 freecell によって指されています(Fig. 5.13).

freecell
id
id
id
セルに関しては以後次のように表す(idの部分の省略)
freecell

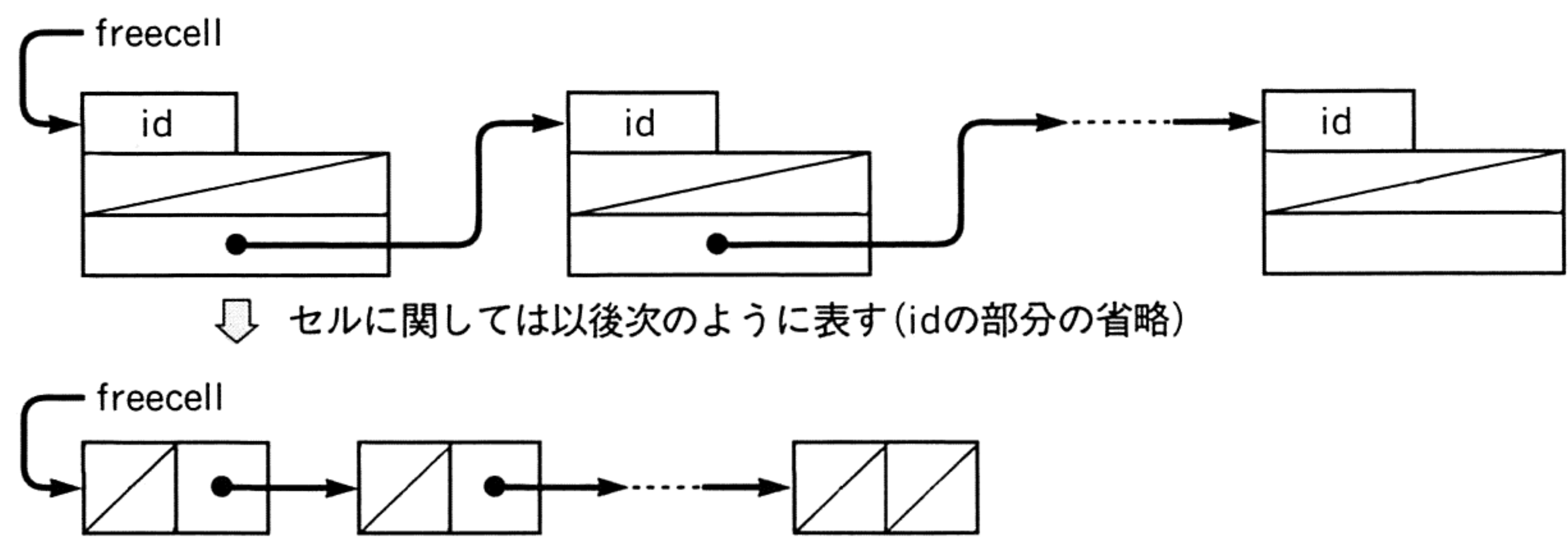



Fig. 5.13 自由リスト

自由リストの各セルの car は nil を指していることに注意してください．したがって，仮に自由リストを印字させたとしたら，次のようになるでしょう．

(nil nil nil nil nil nil nil ・・・・・・・・・ nil) ……nil の数だけアキのセルがある

● newcell() (7行～17行)

自由リストからセルをひとつ切り放します．freecell が nil の時はもうセルがありませんので，エラーとなります．エラーでない時は，それまで freecell が指していたセルの cdr のセルに freecell を移し，今まで自由リストの先頭だったセルの cdr を nil に変えて，そのセルへのポインタを返します(Fig. 5.14).

freecell
freecell
新しく取り出されたセル

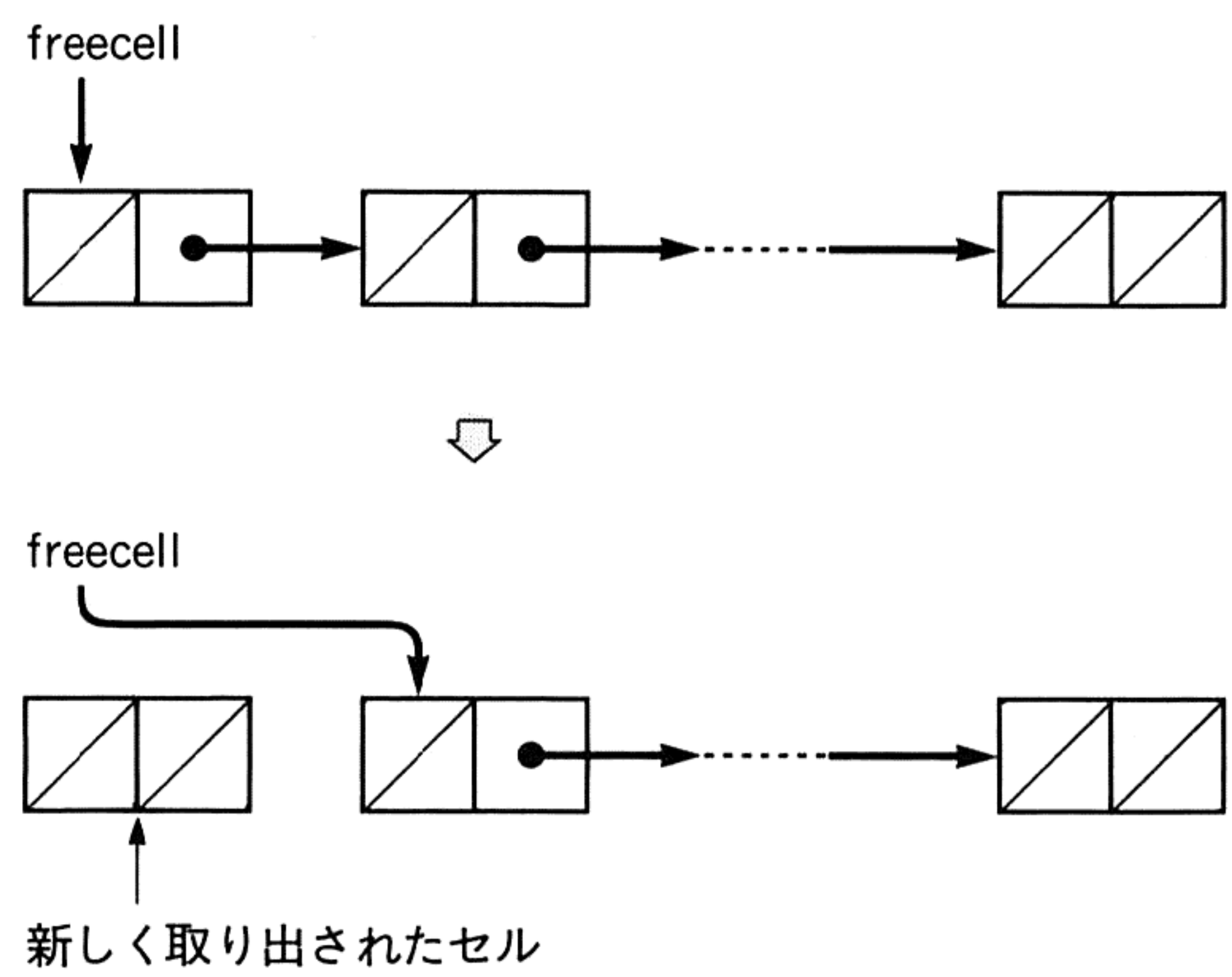



Fig. 5.14 セルの取り出し

5.4.2 シンボルアトムを取り出す

シンボルアトムの自由領域も，セルと同様に直鎖状につなげて管理します．セルの場合は cdr を鎖に使いましたが，アトムの場合は4つのポインタ(属性リスト，値，印字名，関数定義)のうち属性リストへのポインタを使います．つながったアトムの先頭は大域変数 freeatom が指しています(Fig. 5.15)．

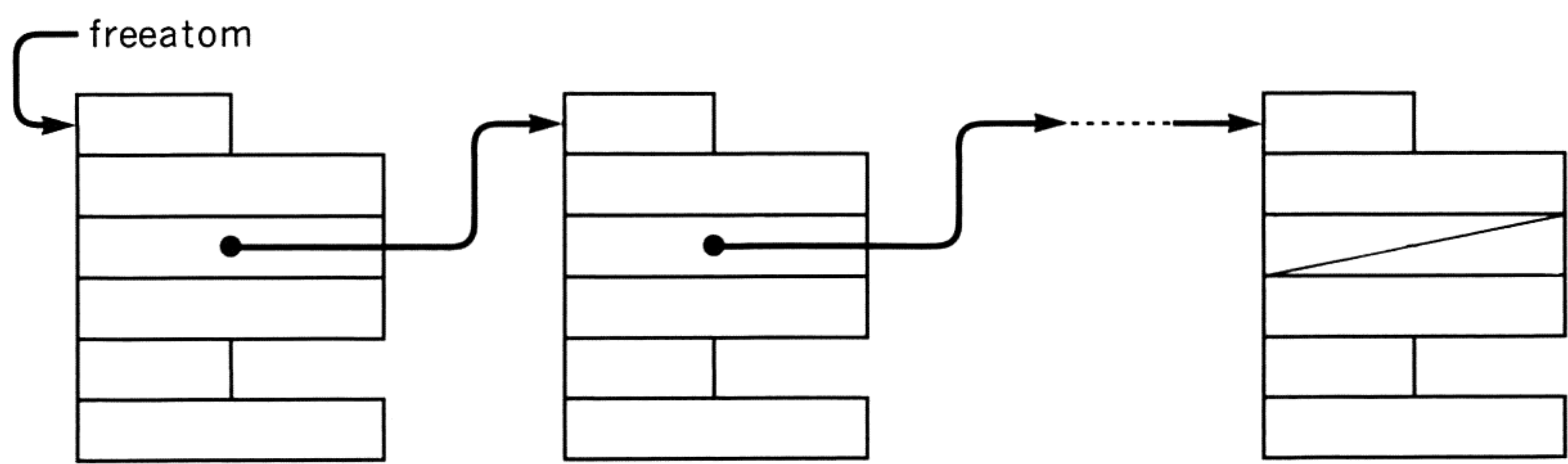



Fig. 5.15 自由シンボルアトム

● newatom() (19行〜29行)

シンボルアトムの切り出しをセルの時と同じ要領でおこないます．ただし，取り出した構造体のメンバへ値をセットはせず，そのまま返します．したがって上位の関数でこれらのメンバの値を整える必要があります．

5.4.3 数値アトムを取り出す

数値アトムの自由領域も，やはりつないで管理します．数値アトムにはこの〝つなぎ〞に使うためのポインタ型のメンバが用意してあります．この数値アトム構造体へのポインタを用いて，セルの時と同様に鎖状につないだものが数値アトムの自由領域となります．自由数値アトムの先頭は，freenum が指しています(Fig. 5.16)．

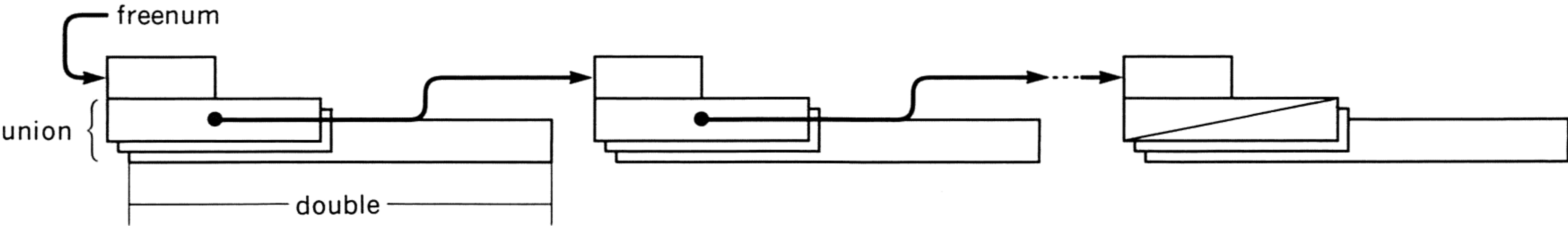



Fig. 5.16 自由数値アトム

● newnum()　(31行～41行)

数値アトムをひとつ切り出します．自由領域の数値アトムのidは，整数型を表す"_FIX"に初期化されているので，この関数から受け取った数値アトムを実数型として使う時は，

np->id = _FLT；

のように，実数型を示す_FLTにidを変更してください．整数型として使う場合は，idの書き換えは必要ありません．

5.4.4　文字列領域の管理

次にシンボルアトムの印字名を格納する文字列領域の使用法について説明します．文字列領域は，このファイルの中で管理するのではなく，read.cファイルの中にあるシンボルアトムを作る関数mk_sub()が直接手を下しています．しかし，後のバージョンにおいては，文字列領域も他の自由領域と同様にガベージコレクタの管理対象となります．

セルと数値アトムとシンボルアトムの3つの自由領域は，上記のように未使用の構造体を数珠つなぎにしてプールし，使用時にはそこからひとつずつ解放する，という形で管理しています．しかし，文字列はこれらのような構造体は持っていません．この領域は先頭から順番に使用され，ある場所から後は全部未使用の形になっています．そして未使用領域の先頭を文字ポインタnewstrが示しています(Fig. 5.17)．

strtop　newstr

n	i	l	¥0	l	a	m	b	d	a	¥0						………

未使用領域

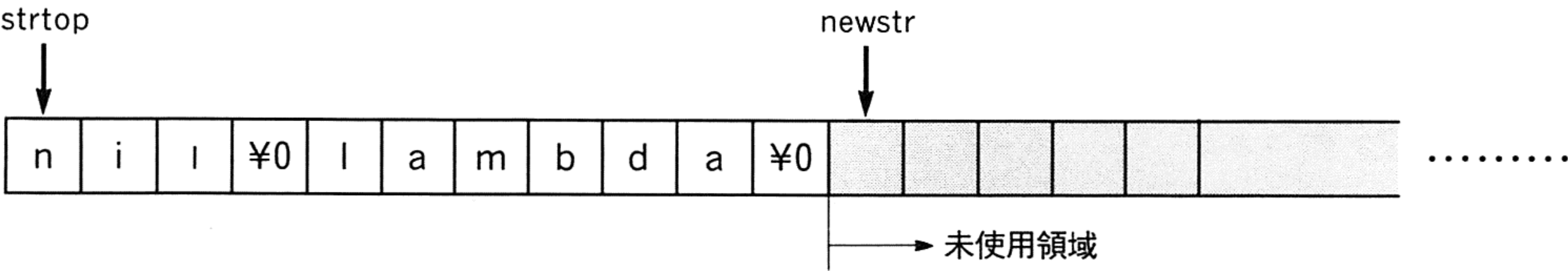



Fig. 5.17　自由文字列領域

5.5 S式の評価　－eval.c－　(List 5.7)

ここまでの作業でようやく，S式を読み込み，リスト構造としてメモリに格納し，それを再び出力することができるようになりました．残るS式の評価の部分が完成すれば，Lisp処理系もほぼ完成です．このS式を評価するための関数をまとめたものがeval.cのファイルです．

5.5.1 S式を評価する

S式を評価するとは，ある環境の下で，次の値を求めることです．

・S式がシンボルアトムならばその値
・S式が数値アトムならばそれ自体
・S式がリストならば関数作用として評価した結果

● eval() （7行〜38行）

S式評価の最も上位に位置する関数がこのeval()です．引数formが評価するべきS式，envが評価に使う環境リストです．

プログラムでは，まず，form->idでformの指す構造体のタイプを参照し，それをキーとしてswitch文で，分岐しています．

formがシンボルアトムの時は関数atomvalue()にformを渡して値を求めさせます．atomvalue()の中でエラーが起こることを考えて，breakでswitch文の次へ進んでいます．

formが数値アトムの時は，その評価値はそのアトム自身ですから，formをそのまま返しています．ここでは何の処理もしていないので，エラーが起こることはありませんから，そのままリターンしてかまいません．

formがセルの時は関数作用とみなして評価します．まずformの先頭要素を関数として取り出し，funcとします．この時，formのcdrは引数のリストになり，argsとして関数applyに渡されることになります．ただし，funcの表す関数が引数を先に評価してから使うタイプ(subr型とexpr型)の時は，引数リストを関数evallist()に渡して各引数を評価してからapplyに渡し，引数を評価しないタイプ(fsubr型)の時は引数リストをそのままapplyに渡します．funcが引数を先に評価するタイプかどうかを調べるには，eval_arg_p()を呼び出しています．関数の実行は，apply()にfuncとargsを渡しておこなわせます．

formの評価が終わり，switch文を抜けたところで，エラーフラグerrを調べます．もしerrの値が〝ERR〟になっていたら，関数pri_errにformを渡してエラーメッセージを出力させ，さらにトップレベルまで脱出を続けるためリターンします．Will o'Lispでは，サブルーチンコールのネストが深いところでエラーが起こると，プログラム中に散在するエラーチェック文〝ec〟によってどんどんネストを抜け出し，いつかはこのeval()にたどりついてエラーメッセージが表示されます．エラーメッセージは，エラーが起きたところに最も近いeval()が表示することになります(「5.6 エラー処理」参照)．

エラーフラグがNULLの場合は，valueに置いてあった評価値を返します．

● eval_arg_p() (40行〜48行)

このeval_arg_p()は，引数funcに渡されてきたS式が，引数を先に評価するタイプのLisp関数かどうかを調べるためのものです．

funcがアトムの時は，funcのアトムの構造体の中からftypeを取り出し，_EAビットの状態を調べます(Fig. 5.18)．_EAビットがonならば，そのfuncは引数を評価するタイプの関数です．

id
value
plist
name
fptr
ftype
M C | E A | S R | U D
マスク _EA 0 0 0 0 0 1 0 0 = 0x04
_UD 0 0 0 0 0 0 0 1 = 0x01

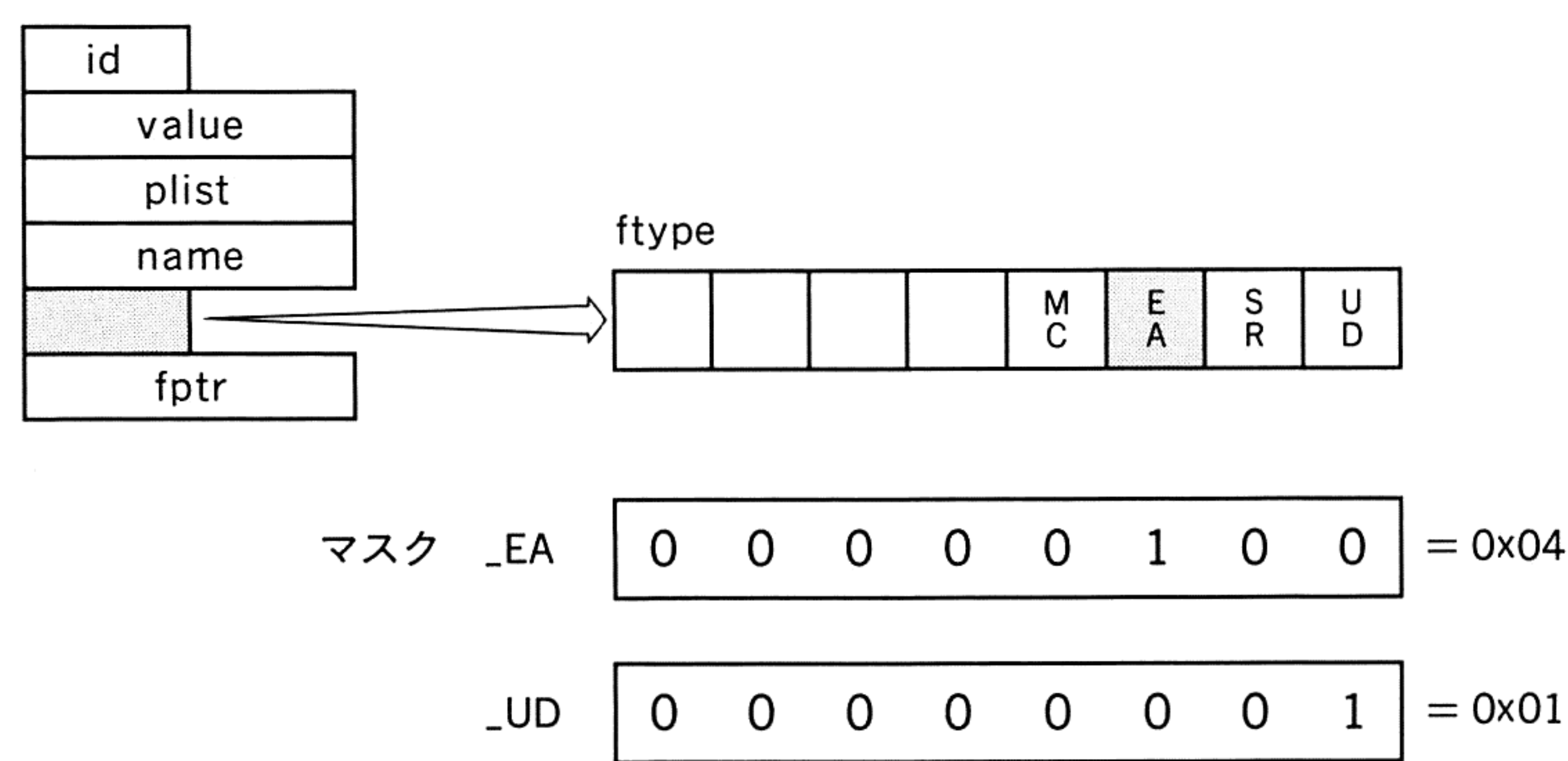



Fig. 5.18 関数タイプの判別

funcがリストならば，リストの先頭要素が〝lambda〟かどうかを調べます．UNDEADバージョンでは，リストで表される関数はlambda式しか存在せず，かならず引数は評価されるのですが，後の拡張では引数を評価しないタイプのnlambda式，macro式なども登場します．

● evallist() (50行〜68行)

与えられた引数リストの各要素を与えられた環境の下で評価し，その結果をリストにまとめて返す関数です．引数argsは評価すべき引数リスト，envが環境リストです．

最初にargsがリストかどうかを確かめ，リストでなければnilを返すようにしています．次に，newcell()を使って新しいセルをひとつ作り，cp1とcp2に置きます．このセルは，評価値をまとめたリストの先頭になるセルで，cp1はこのセルの後ろにセルを継ぎ足してリストを作るのに，cp2は作成されたリストの先頭を押さえておくために使います．

まず，argsの最初の要素を取り出し，eval()に渡して評価させ，結果をcp1のcarに置きます．次にargsをargsのcdrに進めて，そこでargsリストが終わりならばループを抜け出します．argsにまだ要素が残っていれば，cp1の後ろに新たなセルを継ぎ足し，そのセルをあらためてcp1としループの先頭へ戻ります．こうしてargsの要素がなくなったら，できたリストの最後のcdr部にnilを与えて，cp2で押さえておいたリストの先頭へのポインタを返します．

5.5.2 シンボルアトムの値

基本的には，Will o'Lisp のシンボルアトムは，「5.1 データ構造の定義」で述べたようにその構造体自身の中に値を格納する場所を持っています．しかし，関数の引数として使われた場合のように，アトムがローカルな変数として値を与えられた時には，アトム構造体中には値を書き込まずに〝環境リスト〟にその情報を格納します．

環境リストは Fig. 5.19 のようにアトムとその値のドット対をリストとした構造を持っています(なお，このような〝ドット対のリスト〟構造を持つリストを〝連想リスト〟という)．

環境リスト ((x . 3) (mec . (handy handy)))

nil

x 3 mec (handy handy)

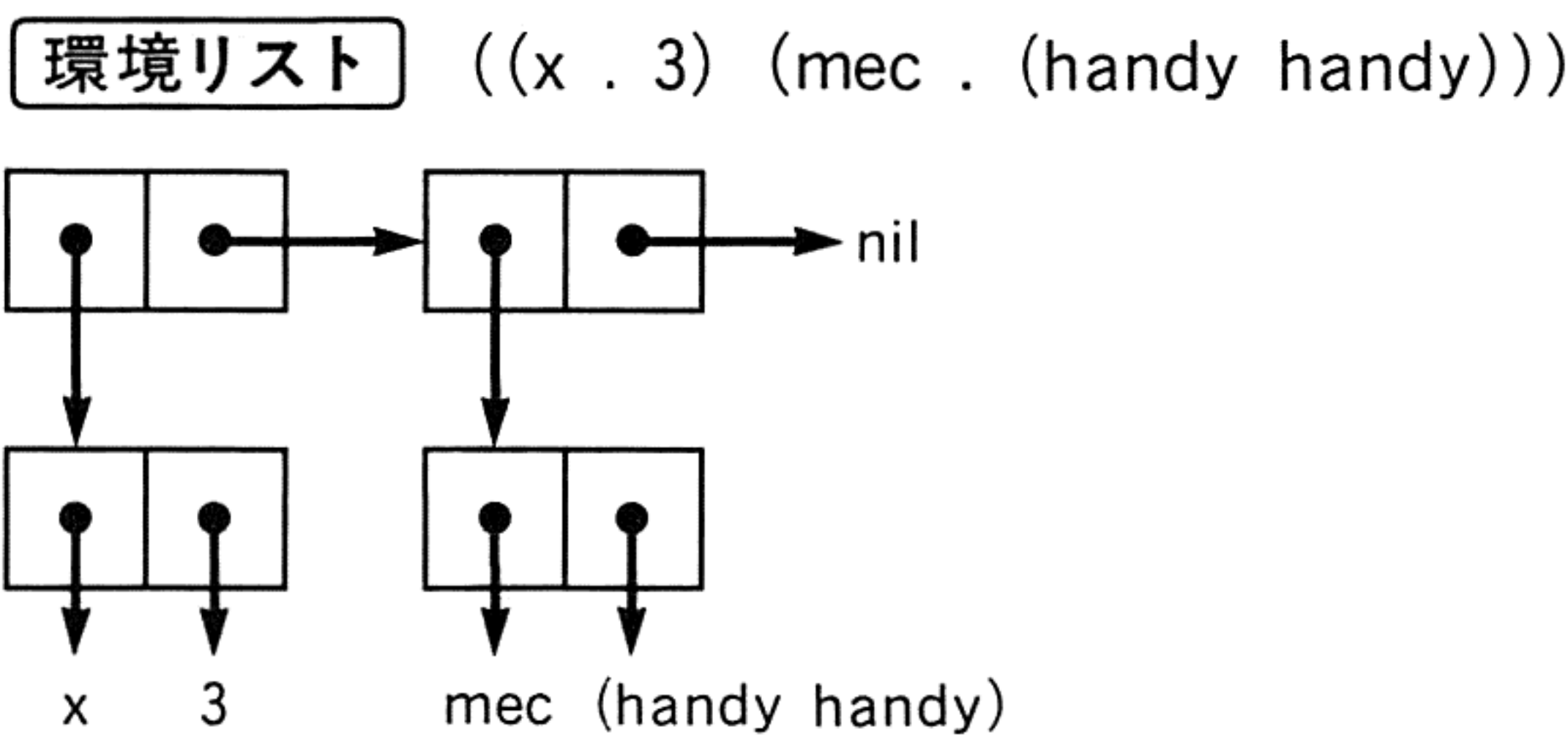



Fig. 5.19 環境リストの構造

シンボルアトムの値が必要とされる時には，まず環境リストが先頭要素から順に検索され，その中に目的のシンボルアトムが存在しなければ，そこで初めてアトム自身の持つ値が採用されます．たとえば，Fig. 5.20 のような環境の下で，アトム pin の値は環境リストから〝4〟となり，アトム zndoko の値は環境リストの中には見つからないので構造体の value メンバを参照して〝3〟となります．また，アトム wool のように環境リスト中に複数回現れているものは，環境リストを先頭から検索していって最初に見つかった値，この場合は〝5〟となります．

アトム自身の持つ値

3 2 0

zndoko pin wool

環境リスト ((wool . 5) (pin . 4) (wool . 4))

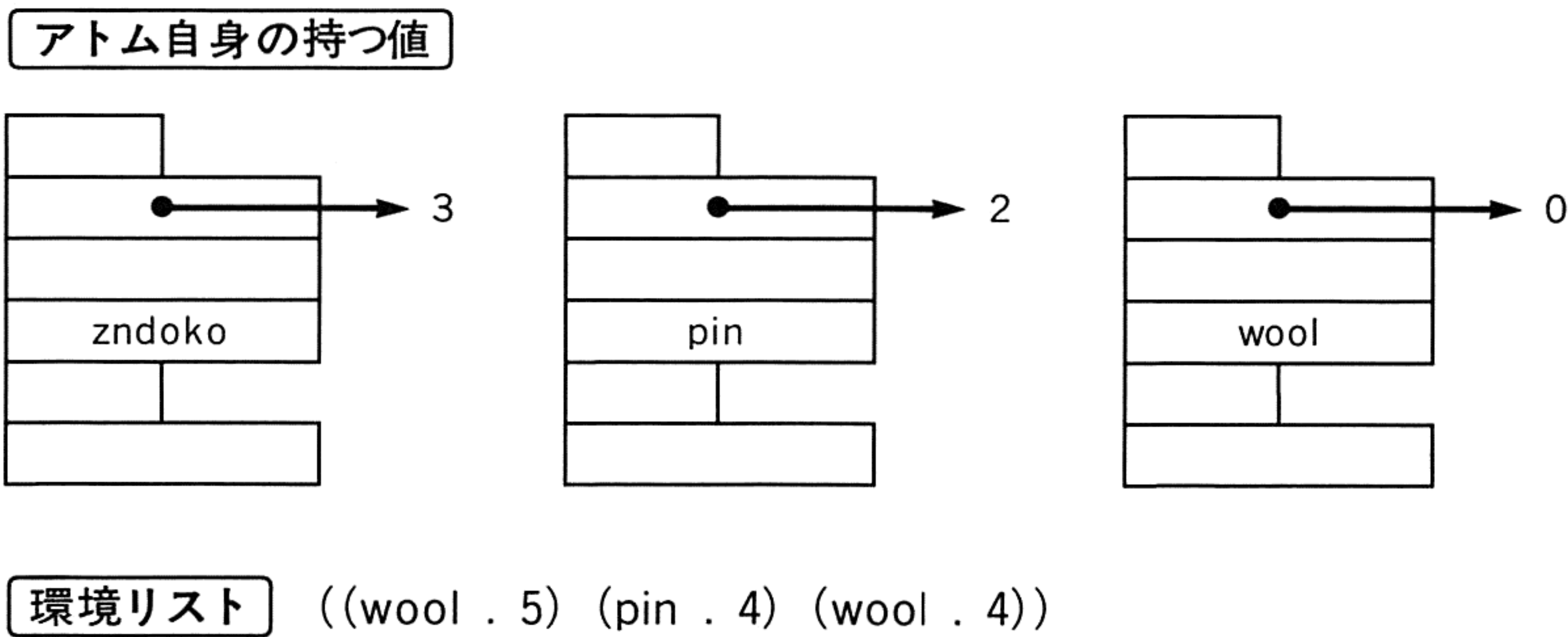



Fig. 5.20 シンボルアトムの値

● atomvalue()　(70行〜84行)

シンボルアトムの値を求める関数です．シンボルアトムへのポインタと任意の環境リストを引数にとります．アトムの値は，lambda-bindされて環境リストの中に存在するか，アトムの構造体の中にメンバvalueとして置かれています．環境リストの中に値がある場合はその値が優先するので，先に環境リストを調べて見つかれば，それを返します．環境の中に見当たらなければ，アトムの中の値を取り出して返します．

プログラムでは，引数apは値を求めたいアトムへのポインタ，envは環境リストです．まず，環境リストの各要素をenv->carで取り出して調べます．環境リストの要素は，かならずセルのはずなので，もしアトムの要素があったら〝Environment has an Atom〟エラーを返します．要素がセルならば，そのcar，すなわちenv->car->carとapを比べ，一致したら要素のcdrをそのアトムの値として返します．一致しなければ，環境リストの次の要素へ移り，ループの先頭へ戻ります．環境リストの中にapに一致するものがなければ，アトムの構造体の中にある値をap->valueとして取り出し，これを返します．

5.5.3 関数作用の評価

評価すべきS式がリストであれば，それを関数作用として評価せねばなりません．ここで，〝リストを関数作用として評価する〟とは，そのリストが，

(<関数> <引数1> <引数2> …… <引数n>)

という形であるとして，関数を引数に適用して結果を求めることです．Will o'LispのUNDEADバージョンでは，〝関数〟の形態として3つの種類があり，それぞれ次のような処理をおこないます．

(1) アトムで表されていて，アトムの構造体中のfptrメンバにCのオブジェクトコードによる関数定義へのポインタを持つもの．この場合は，そのオブジェクトコードをコールします．

(2) アトムで表されていて，アトムの構造体中のfptrメンバにLispによる定義(lambda式)へのポインタを持つもの．この場合は，lambda式を取り出し，そのlambda式を用いて(3)の処理をおこないます．

(3) lambda式．ただしlambda式とはFig. 5.21のようなリストです．この場合は，まずlambda式の中の仮引数と実引数を結び付け，新しい環境を作り出します(lambda binding)．そして，その環境の下で本体を順に評価し，最後の本体を評価した結果を〝関数を引数に適用した結果〟とします．

(lambda 〈引数リスト〉〈本体1〉……〈本体n〉)

例 (lambda (x y) (cons y x))

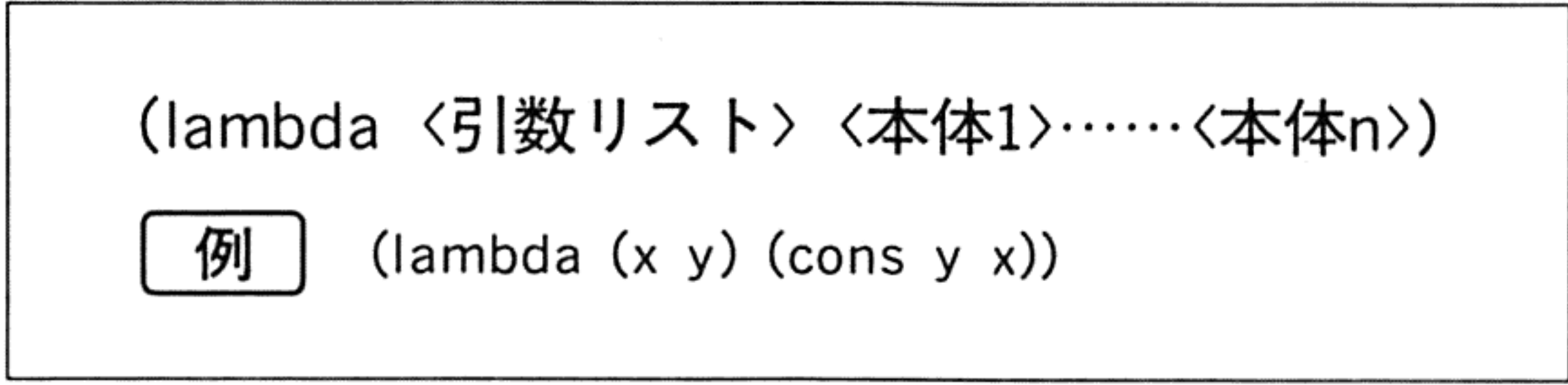



Fig. 5.21 lambda 式

● apply() (86 行～120 行)

apply()の役割は，与えられた環境 env の下で，関数 func を引数 args に適用し，値を求めることです．

プログラムでは，まず func->id を参照して関数が上記の(1)～(3)のどの形態で apply()に渡されてきたのかを調べ，それをキーとして switch 文により分岐します．

まず，関数がアトムの時は，そのアトムの構造体から関数のタイプを示すフラグ(ftype)を取り出します．このフラグはビットごとに意味づけられているので，マスクと and をとって状態を取り出します．

_UD ビットが on になっている時は，そのアトムを名前とする関数は定義されていないことを意味するので，〝Undefined function〟エラーを返します．

_SR ビットが on になっている時は，その関数の定義が C のオブジェクトコードで与えられていることを意味します．この時は，アトムからオブジェクトコードへのポインタを取り出してコールします．このコールのやり方を次に説明します．

C の変数は，関数へのポインタという型をとることができます．たとえば次のプログラムリストを見てください．

```
1 : #include  <stdio.h>
2 : main()
3 : {
4 :         int  printf() ;
5 :         int  (* foo)() ;
6 :
7 :         foo = printf ;
8 :         (* foo)(" hellow ! ! ¥n") ;
9 : }
```

5 行目で，変数 foo は〝int の値を返す関数へのポインタ〟として宣言されています．この foo には，7 行目でおこなっているように，int を返す任意の関数へのポインタが代入可能です．また，foo が指している関数を実行するには，8 行目のように〝(* foo)〟でそのポインタの先にある関

数の実体を取り出して，引数を与えてやればよいのです．

さて，同様に考えて，〝(セルへのポインタを返す関数)へのポインタ〞は，

```
CELLP  (* foo)();
```

として宣言することができます．Will o'Lisp では，C で関数定義が与えられたアトムの中には，〝fptr〞というメンバ名でこの型のポインタがかならず入っていますから，それを取り出してきて実行すればよいわけです．取り出したポインタを格納しておくために，〝funcp〞という変数を用いることにすれば，これは次のように書けるでしょう．

```
CELLP  (* funcp)();          /*  funcpの宣言  */
  ......
funcp  =  func->fptr;
return  (* funcp)(args);
  ......
```

ただし，アトムの構造体宣言をする時に fptr は CELLP 型として宣言してあるので，これを funcp へ代入する際にはキャストをかけて型を合わせなければなりません．キャストは，一般に，変換して得ようとする型の型宣言から変数名を取り除いて得ることができます．したがって，ここでのキャストは，

```
(CELLP (*)())
```

となります．

また，apply()に渡されてくる時引数 func は CELLP 型に宣言されていますから，まず(ATOMP)のキャストをかけておかないと，fptr メンバを取り出すことができません．

結局，関数へのポインタを取り出すためには，

```
funcp = (CELLP (*)())((ATOMP)func)->fptr;
```

というように，func にキャストをかけてアトムへのポインタに変え，そのアトムの fptr の値を取り出し，さらにそれにキャストをかけて関数へのポインタを生成する，という作業が必要になります．

funcp を取り出した後の if 文はこの関数が引数を評価するタイプかどうかを調べて，環境を引数リストと一緒に渡してやるかどうかを決めています．

_EA ビットが on になっている関数は引数があらかじめ評価してから渡されるタイプ(subr)であり，自分の中で評価をおこなうことはありません．したがって環境も必要ではありません．

一方，引数をあらかじめ評価しないタイプ(fsubr)の関数では，関数内部でいくつかの引数を

評価することがあります．たとえば，*setq* は，第1引数(および奇数番目の引数)は，評価されませんが，第2引数(および偶数番目の引数)は，評価されなければなりません．このため，setq_f()は，その中で eval()を呼んでいます．この時環境が必要になるので，fsubr 関数を呼ぶ時には環境を一緒に渡してやる必要があるわけです．

funtype の_SR ビットが立っていなかった場合は，関数 func は Lisp による関数定義，すなわち lambda 式を持っています．その時は，この lambda 式を取り出してあらためて func とし，そのまま次の〝func が lambda 式の時の処理〟へ進みます．

関数 func がリストの時は，そのリストが2つ以上の要素を持っているかどうかを調べます(func の cdr を調べ，それがセルであれば2つ以上の要素を持つことがわかる)．要素が少なくとも2つ存在しなければ正しい lambda 式ではあり得ないので，〝IFF(Illegal function form)〟エラーを返します．

ここで，関数 func の第1要素(すなわち car)が〝lambda〟ならば，func は Fig. 5.21 のような lambda 式であり，その処理に入ります．この if 文の条件式の中で，lambda に(CELLP)というキャストがかかっているのは，比較の対象である func->car がセルへのポインタだからです(セル構造体のメンバ car，cdr は，ともにセルへのポインタとして宣言されている)．もし func の第1要素が〝lambda〟でなければ正しい関数ではないので，default 文へ抜けて，IFF エラーを返します．

lambda 式の処理では，まず関数 func の3要素目以降(すなわち cdr の cdr)にある関数の動作本体を取り出し，bodies とします．次に func の第2要素(すなわち cdr の car)をとって仮引数リストを取り出し，実引数リスト，環境と一緒に関数 bind()に渡して lambda-bind させます．bind()は，lambda-bind が済んだ状態の新しい環境を返すので，これを cp に置きます．その次の for 文で bodies から本体をひとつずつ取り出し，cp に置いた環境の下で eval()に評価させます．eval()の結果は毎回 result に置くので，最後に result に残った値が最後の本体の評価値になります．

5.5.4 引数の binding

Will o'Lisp がおこなう lambda-bind(実引数を仮引数にセットする操作)の形式には3種類あります．

(1)仮引数が通常のリストとして与えられた場合

仮引数リストと実引数リストの先頭から，仮引数と実引数を取り出して，順に bind します．もし仮引数リストが空にならないうちに実引数がなくなったら引数の数が足りないとして，"Not enough arguments"エラーを起こします．一方，仮引数が先になくなったら残りの実引数は無視します．

仮引数リスト　：　(x y z)
実引数リスト　：　(a b c)

この場合は，x と a，y と b，z と c をそれぞれ bind します．

(2)仮引数リストの最後にドット対がある場合

仮引数リストの最後のセルの cdr が nil 以外のアトムならば，そのアトムに，その時点で残っている実引数のリストを bind します．

仮引数リスト　：　(x . y)
実引数リスト　：　(a b c)

この場合は，x と a，y と(b c)をそれぞれ bind します．

(3)仮引数がアトムの場合

そのアトムと実引数リストを bind します．

仮引数リスト　：　x
実引数リスト　：　(a b c)

この場合は，x と(a b c)全体を bind します．

● **bind()**　(122行～142行)

プログラムでは，keys が仮引数リスト，values が実引数リスト，env が環境リストです．

まず keys が nil でないアトムかどうかを調べ，もしそうなら，関数 push()を使ってそのアトムと values を環境リストに push してこれを返します．push は環境リストの先頭に，仮引数と実引数のドット対を付け加える操作です．

keys がリストならば，keys および values の要素を順に取り出し，環境リストに push していきます．もし先に values の要素がなくなったら，"Not enough arguments"エラーを返します．仮引数リストの最後のセルに到達したら while を抜け，そのセルの cdr が nil ではないアトムかどうかを調べます．もし nil 以外のアトムなら残りの values とそのアトムを push します．

最後に，得られた新しい環境リスト env を返します．

● push() (144行～156行)

与えられた仮引数と実引数をconsしてドット対とし，これをさらに与えられた環境リストの先頭に付け加えて返す関数です．keyが仮引数，valueが実引数，envが環境リストをとります．プログラムでは，先に環境に新しいセルをつなぎ，そこへドット対のセルをつないでいます．まず，cpに新しいセルを置き，そのcdrにenvをつないで，さらにenvをcpで置き換えます．envのcarには別のセルをつなぎ，このセルのcarとcdrにそれぞれkeyとvalueを置き，envを返します．

5.6 エラー処理 —error.c— (List 5.8)

エラー処理をおこなう関数と，エラーメッセージの配列を含むファイルがerror.cです．ここでは，このファイルに含まれる関数，およびWill o'Lispのエラー処理機構について説明します．

5.6.1 サブルーチンのネストと大域脱出

プログラムは一般に何層ものサブルーチンコールのネストになっています．すなわちメインルーチンがサブルーチンAをコールし，サブルーチンAがサブルーチンBを呼び出し，サブルーチンBがサブルーチンCを……，という形になっているのが普通です．

この時，A,B,Cが異なるサブルーチンであるとは限りません．たとえば，BがコールしているCが実はBそのものであるような場合もありえます．このような自分自身を呼び出す関数呼び出しを〝再帰的呼び出し〞といいます．直接BがBをコールする場合だけでなく，AがコールしたBの中でAがコールされるような間接的な再帰的呼び出しもあります．Will o'Lispは，再帰的呼び出しを多用しています．一般に再帰的呼び出しを含むプログラムは，サブルーチンコールのネストがかなり深くなります．

サブルーチンコールのネストが深いところでエラーが起きた時，その場で処理できる簡単なエラーならともかく，通常はエラーによる破綻を回復するためにトップレベルのルーチンに戻って諸条件を整え直さなければ処理を再開できません．つまり，イニシャライズをやり直すわけで，そのためにはできるだけ速くネストを抜ける必要があります(Fig. 5.22)．

これは，自然なプログラムの流れから見ると例外的な処理であり，うまく書かないとプログラムの美しさを損なうことになります．問題は，エラーが起きたという情報をいかに伝達するかにあります．

func1()　func2()　func3()
func2()　func3()　if(……)
return(error(…));
ec;　ec;

―― 通常処理
-------- エラー検出時の処理

```
CELLP func1(……)
{
  ……
    func2(……); ec;
  ……
}

CELLP func2(……)
{
  ……
    result = func3(); ec;
  ……
}

CELLP func3(……)
{
  ……
    if(……) return(error(……));
  ……
}
```

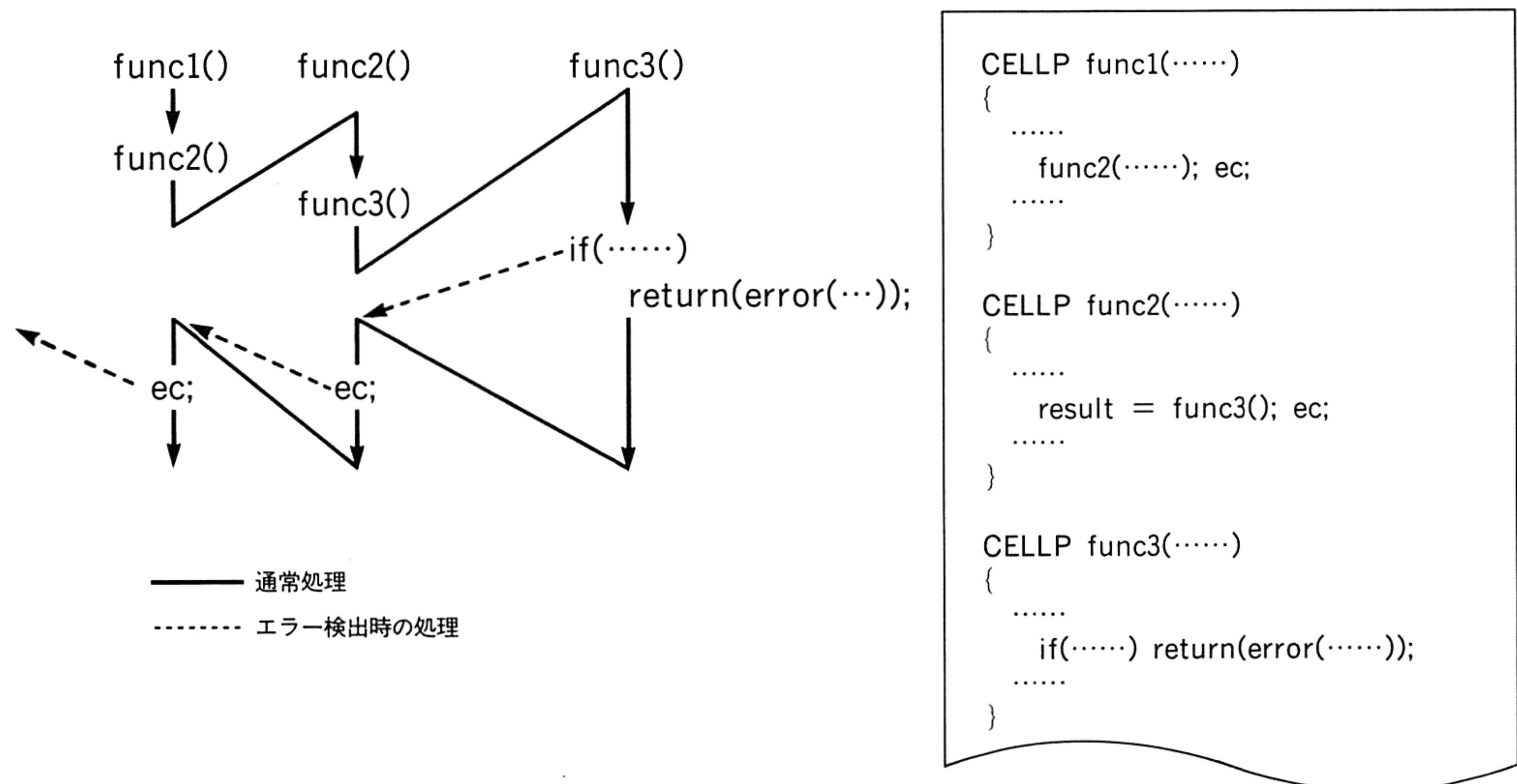



Fig. 5.22　エラー時の例外的な処理の流れ

エラーの発生を呼び出し側の関数に伝える方法としてよく使われるのは，正常な処理が行われた場合には絶対にあり得ない値を返すことです．たとえば，Cの関数fopen()は正常にファイルをオープンできた時にはファイルポインタを返しますが，オープンに失敗した場合，NULLポインタを返すことで異常を呼び出し側に伝えます．

```
FILE *fp;
char *filename;
  ……
if (! (fp = fopen (filename, "r"))) {
  fprintf (stderr, "can't open ¥n");
  exit (1);
}
```

しかし，fopen()のようにプログラム中にたかだか数回しか現れないような関数のエラーならば，このように関数から返ってきた値をいちいちチェックしてもよいのですが，Will o'Lispでは関数を呼び出すごとに至る所でエラーチェックをおこなわねばなりません．そのため関数呼び出しをすべてif文の中に書いていたのでは，本来の処理の流れが見えにくくなってしまい，うまくありません．

もうひとつの方法は，エラーの発生を示すフラグを大域変数としてどの関数からでも参照できるようにしておくことです．エラーを検出した関数ではエラーフラグをonにしてからリターンすることにしておきます．そして，どのような処理をおこなっていても，何らかの関数がコールされた後にはかならずエラーフラグを調べ，onになっていればそれ以降の処理を中断してリターン

することにします．こうすると，1度エラーフラグがonになれば通常の処理はすべてキャンセルされ，ひたすらトップレベルへ戻っていくことができます．この方法を用いると，関数のコールとエラーチェックのif文が分離できます．また，エラーチェックの表現がすべて同じになります．たとえば，エラーフラグの名前をerrとすると，

```
if (err) return;
```

という文をすべての関数呼び出しの後に書いておけばよいわけです．

5.6.2 Will o'Lisp の大域脱出

Will o'Lisp では，上で説明したエラーフラグを用いる方法でエラー処理を実現しています．大域変数errの値は通常は〝NONERR(=0)〟に設定されていますが，関数がエラーを検出するとその値を〝ERR(=1)〟に変更してリターンします，またその関数を呼んだ側では，errの値を参照してエラーの有無を確認し，処理を先に進めるかどうかを判断します．

● error()（11行～17行）

エラー発生の現場で，応急手当(エラーフラグとエラー番号の設定)をする関数です．エラーを検出した関数は，エラー番号を引数として与えてこの関数をコールします．エラーナンバーは，ファイルlisp.hの中で通常アルファベット3文字のエラーコードとして定義されています．

error()はこのようにエラー番号を渡されると，まず大域変数のエラーフラグerrに〝ERR〟を代入して，〝エラーが発生し，エラーメッセージはまだ出力されていない〟という状態を表します．次にerr_no(これも大域変数)にエラー番号を代入します．2つの作業を終えると，error()はNULLを返します．

エラーを検出した関数は，もうそれ以降の処理を実行する必要はありませんから，すぐにリターンしてかまいません．その際に戻り値としていちばん無難なのはNULLを返すことです．Will o'Lisp では，error()を呼ぶ関数はCELLP型かATOMP型またはNUMP型ですが，NULLポインタを返しておけば(NULLポインタの型は不定なので)コンパイラに〝Return value type mismatch〟とか何とか文句をいわれずにすみます．したがって，エラー検出時には次のようにerror()から戻ってきた値をそのまま返せばよいわけです．

```
CELLP func()
  ......
  return error( エラー番号 );
  ......
```

●エラーチェックマクロ　ec

error()がセットしたエラー情報の伝達をおこなうため，エラーを検出する可能性のある関数をコールしたら，そのすぐ後ろに〝ec；〟というマクロをおくようにします．

エラーを検出する可能性のある関数とは，その中あるいはそれより下位の関数の中でエラーフラグがセットされる可能性のある関数，具体的にいえばerror()またはecを含んでいるようなすべての関数です．

ecは〝error check〟の略で，ファイルlisp.hの中で

```
#define      ec      if(err)return(NULL)
```

とマクロ定義してあります．このecを各関数呼び出しの直後に配置しておけば，いったんエラーフラグerrがセットされるとreturnが連鎖的に起こり，結果として大域脱出が実現されます．このようにエラーチェックを〝ec〟の二文字に詰め込むことにより，エラーチェックがほとんど目立たなくなって，プログラム本来の処理の流れを例外処理に妨げられることなく表現できます．

ところで，プログラムのどの場所からも利用できる大域変数は，うまく使用すれば，情報伝達の手段として，たいへん重要な役割を果たすことができます．UNDEADバージョンでは，errがエラー発生時にしか使われていませんが，以降の拡張バージョンではerrはさらにさまざまな役割を持たされ働くことになります．

5.6.3 エラーの表示

こうして下位のルーチンから抜け出してきたエラーは，どこかで捕まえて表示してやらねばなりません．エラーメッセージの表示はどこでやってもかまわないのですが，前述のように，Will o' Lispでは関数eval()の中にエラーメッセージ表示ルーチンを置き，エラーを表示するようにしています．

たとえば，次の①のＳ式を評価することを考えましょう．

(plus (plus 'a 1) 2) ……①

①のＳ式をeval()に渡すと，*plus*が引数を評価するタイプのsubr関数ですから，まず引数を評価しなくてはなりません．そこで，第１引数の

(plus 'a 1) ……②

を評価するために，eval()が再帰的に呼び出されます．ところが，この引数〝'a〟は数値ではありません．そこでplusはエラーを検出し，error(IAN)を実行してエラーフラグerrとエラー番号

err_no をセットした後リターンします(IAN は〝Illegal argument -- Number required〞の略).

すると，ec の働きにより大域脱出が起こってサブルーチンのネストを次々と抜け出し，やがて②のＳ式を評価しようとしていた eval()まで戻ってきます．eval()はエラーメッセージと自分が評価しようとしたＳ式(ここでは〝(plus 'a 1)〞)を表示することになります．

● err_msg [] (19 行〜48 行)

エラーメッセージへのポインタが登録されている配列です．ファイル lisp.h で定義されているエラーナンバーは，対応するエラーメッセージがこの配列中の何番目にあるかを示しています．たとえば，〝Not enough arguments〞は，err_msg 配列中，10 番目にあるので，これに対応するエラーコード〝NEA〞は lisp.h の中で 10 に定義されています．

● pri_err() (50 行〜64 行)

エラーメッセージの表示を実際におこなうのは関数 pri_err()であり，eval()が pri_err()にエラー発生箇所を示すＳ式を渡してコールします．すると pri_err()は，erro_no を見てエラーメッセージを表示し，さらに渡されたＳ式を表示します．

プログラムの前半では fprintf()でこれらを順に出力しています．〝At 〞を出力した後の if 文は，トップレベルで起こったエラーを見分けてエラー発生箇所の表示を toplevel とするためのものです．トップレベルでＳ式を読み込む際に起きたエラーはまだ eval()を通っていないため，エラー発生箇所を表示するためにエラー情報を捕まえてくれるものがありません．この場合は，eval()の代わりに最上位の main()関数が pri_err()をコールします．その時 pri_err()には nil が渡されます．したがって，pri_err()では渡された引数が nil だったら，トップレベルでのエラーと判断します．それ以外の一般のエラーは，引数をそのまま出力するようにしています．

さて，pri_err()がエラーを表示しても，実はまだエラー処理は終わりではありません．文字バッファの初期化や入力ストリームの再設定など，たくさんの作業が残っています．この処理をおこなうためには，もっと上位のルーチンへ戻る必要があります．

そのために，再び大域変数 err が用いられます．pri_error()は，エラーを表示した後，エラーフラグ err の値を〝ERROK〞，つまり〝エラーが起きたが，メッセージはすでに表示した〞という状態に変えてリターンするのです．ERROK の値は 0 ではない(−1 である)ので，この状態で ec に出会うとやはり関数からの脱出が起きます．したがって，再び ec によってどんどんネストを抜けていって，プログラムの流れはトップレベルまで戻ることになります．この途中で eval()に出会っても，エラーフラグの値がすでに〝ERR〞ではなくなっているため，もうエラーを表示することはありません．

トップレベルに戻った後の処理については，次の main.c の説明で述べることにしましょう．

5.7 トップレベルループ －main.c－ (List 5.9)

Lisp インタープリタの作業は,

- S 式を読み込む.
- 読み込んだ S 式を評価する.
- 評価した結果の S 式を表示する.

の繰り返しです. それぞれの機能の実現についてはこれまでに述べてきました. それらを組み合わせてループを作れば Lisp 処理系ができてしまうのです. 何と簡単なことではありませんか. もちろん, この後たくさんの組み込み関数を作らねばなりません. しかしここまでくると, 百里の道の九十里はきたような気分になり, 〝完成間近〟という言葉が自然と出てくるのも無理からぬことといえましょう.

このトップレベルループと, その他もろもろの初期設定をおこなう関数を集めたものが main.c のファイルです.

5.7.1 トップレベルループの処理

Lisp でトップレベルというのは普通 read-eval-print ループを指しています. しかし実際の処理系では, このさらに外側にもうひとつ, 入出力の初期化やエラーの処理をおこなうループが存在しています. Will o'Lisp では, それを main()がおこない, read-eval-print ループを toplevel_f()が処理しています.

● main() (12 行〜22 行)

greeting()によってブート時のメッセージを出した後, init()でシステムの初期化をします. 初期化の後無限ループに入ります. この中身は入出力関係の設定とトップレベルループの呼び出し, そしてエラーの処理です.

ループの最初に, reset_err()を呼び出して, 以下の設定をおこないます.

- 現在の入出力先を, 標準入出力に設定する.
- エラーフラグ err を NONERR にセットする.
- S 式入力用バッファのポインタを先頭に戻し, バッファを空にする.

次に toplevel_f()を呼び出します. 通常はこの toplevel_()の中で Lisp インタプリタの処理が繰

り返されていますが，エラーが発生すると処理の流れはmain()に戻ってきます．この時，前述のようにトップレベルでのS式読み込みの途中にエラーが起こってしまった場合は，main()に戻ってきてもエラーメッセージが未表示のままです．しかし，すでにエラーメッセージが出力されていれば，エラーフラグerrは〝メッセージ出力済み〟を示すERROKになっているはずです．そこで，errを調べてもしERROKでなければ，pri_err()を呼び出してエラーメッセージを表示することにします．

また，toplevel_f()の中で〝EOF〟が読み込まれた場合にも，main()に戻ってきます．その場合に備えて，標準入力を再設定する必要があります．この処理はreset_stdin()が実行します．

● toplevel_f()　(24行～44行)

Lispのトップレベルの動作の実体がここにあります．すなわち，リーダ→エバリュエータ→プリンタの無限ループです．このループから脱出するための条件は2つあり，いずれか片方が満たされればこの関数から抜け出します．ひとつは内部でエラーが起きること，もうひとつはリーダがEOFを検出することです．

本プログラム中では，〝_f〟の付いたCの関数はLispの関数の定義であるという原則があります．しかしながら，このtoplevel_f()に対応するLispの関数は存在しません．それは，このtoplevel_f()がLispの関数として登録されても差支えない形をしてはいるものの，本書の範囲内では有効な使い道が見つからないからなのです．環境を引数に加えて〝ローカル〟トップレベルを作るとか，内部のリーダやエバリュエータやプリンタを表す関数名までも引数にしておいて，それらのすり替えを可能にする(hookという)など，面白いことができると思うのですが，それは今後の課題といえましょう．

5.7.2　初期化

● init()　(55行～103行)

システムの初期化をおこないます．初期化には領域に関するものとシンボルアトムに関するものの2つがあります．ここには領域に関するイニシャライズの処理があります．

まず，malloc()で領域を確保しますが，この時に引数の

sizeof(○○) ＊ ○○SIZ

が符号なし整数の最大値を越えないように十分注意して，

○○SIZ

を設定してください．オーバーフローが起こると要求した領域と確保された領域との大きさが合

わなくなり，誤動作を起こします．メモリーが足りなくてどれかひとつでも領域が確保できなかった時(malloc()がNULLを返した時)は，メッセージを出してOSへ戻ります．

次に確保した領域から自由リストを作りますが，その時注意することがひとつだけあります．それは，この作業の中でnilへのポインタが必要になるということです．ですから，nilというシンボルアトムが何処にあるのか少なくともその場所だけは確定していなくてはなりません．そこで，シンボルアトム領域の先頭の構造体をnilに割り当て，シンボルアトム領域の2番目からを自由領域として使います(Fig. 5.23)．自由領域の構造はそれぞれの構造体をポインタで順番につなぎ，最後をnilで閉じたものです(「5.4 自由領域の管理」参照)．

atomtop
freeatom
nil
nilには
手を触れない
これ以降を初期化する

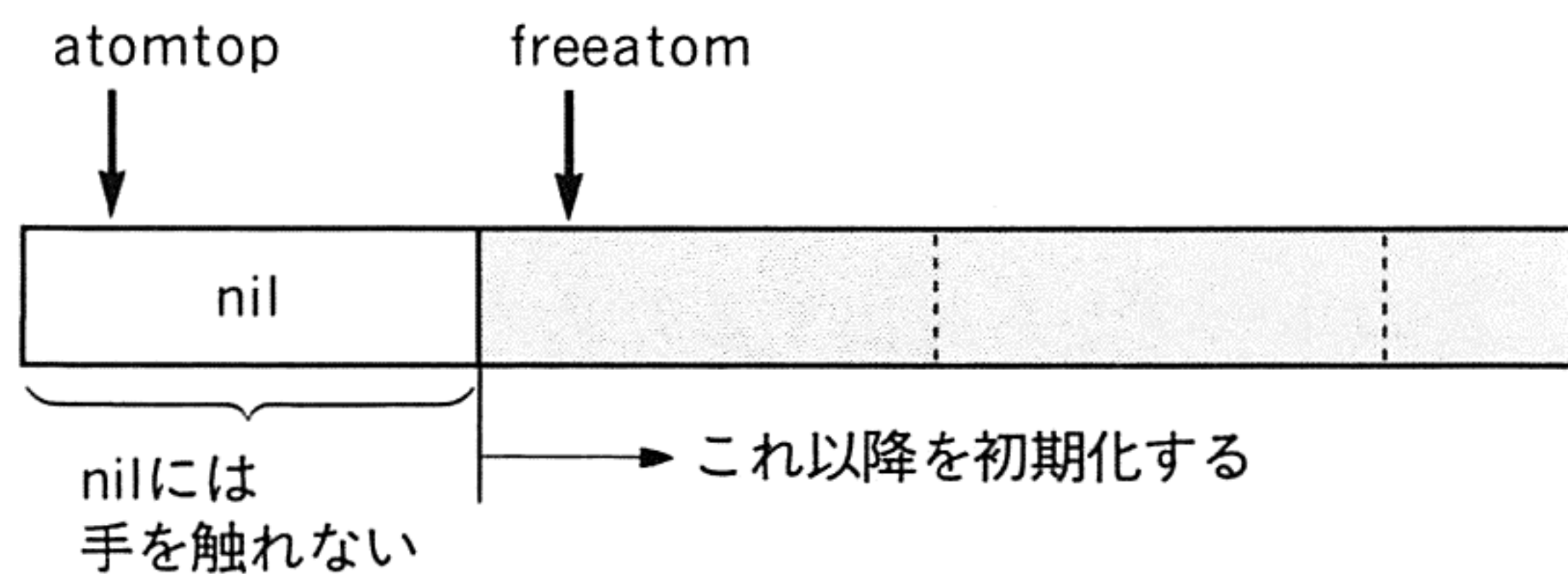



Fig. 5.23　自由シンボルアトム領域の初期化

領域の初期化の最後としてoblistのテーブルを初期化します．この時点ではまだひとつもシンボルアトムは存在していませんから，すべてのキーのテーブルをnilにしておくだけです．

このあと，いくつかの基本的なシンボルアトム(システムアトム)と関数の初期化をおこないます．関数の初期化については「5.8 関数の作成」で後述しますので，ここでは残るシステムアトムの初期化に触れておきます．

● mk_nil()　(105行〜118行)

初期化に先立ち，nilの構造体のメンバをセットします．先に述べたとおりnilはその存在位置が決まっているので，汎用のシンボルアトムを作る関数は使えません．それで，この関数でnilの中身を整えます．文字領域に印字名をコピーし，メンバを整え，oblistに登録します．

● mk_sys_atoms()　(120行〜129行)

nilほど特別ではないが，インタープリタがそのポインタを知っておくべきいくつかのシンボルアトムがあります．これら〝t〞や〝lambda〞などのシンボルアトムはここで汎用の関数を用いて作られます．それはリーダが新しいシンボルアトムを読み込んだ時と同じ手順です．すなわち，未使用のシンボルアトム領域から構造体を切り出し，中身を適当に整え，oblistに登録するのです．この機能はリーダ内の関数mk_atom()が持っているので，それを呼べばよいだけです．

5.8 関数の作成

ここでは，subr 型，fsubr 型の Lisp 関数を C によって定義しています．C の関数の名前は，対応する Lisp 関数の名前に〝_f〟をつけたものになっています．したがって，その機能は，Lisp のマニュアルを見ればわかるようになっているので，細かくは説明しません．ここでは，Lisp 関数を C で定義する時の一般的形式を述べていきます．

5.8.1 関数記述の規則

たとえば，*rogue* という Lisp の関数を作りたいとしましょう．約束として C の関数の名前は対応する Lisp の関数の名前に〝_f〟を付けたものを使うことにします．*rogue* の場合は rogue_f です．

Lisp の関数はすべて S 式を返します．Will o'Lisp では，S 式というデータ型はセルで代表することにしてあるので，すべての関数は CELLP(セルへのポインタ)型の関数として宣言します．たとえその関数がシンボルアトムしか返さないとか，数値しか返さないことが明白であっても，CELLP 型と宣言しておけば，後でよけいな〝type mismatch〟の類の警告をコンパイラに出されずに済みます．

引数は一律に args という CELLP 型の変数にします．*apply* がこれらの関数を呼び出す時に，この args に引数をまとめたリストを，subr 型の場合は評価して，fsubr 型の場合は評価せずに渡すようになっています．受け取った側では，そのリストから再びそれぞれの引数を取り出すのです．

しかしその前に，引数の数をチェックする必要があります．本書に掲載した Will o'Lisp の関数では，多すぎる引数にはエラーを出していません(ひとの好意は素直に受け取る)が，引数の不足はきびしくチェックします．任意個数の引数をとる関数でも，最少限必要な数に足りているかどうかを調べます．渡された引数の数は，すなわち args が指しているリストの長さです．リストの長さは cdr を取っていって何回目に nil にぶつかるかでわかりますが，ドット対のことを考えると，リストの終わりは nil かどうかで見るよりセルでないかどうかで見た方が確かです．

たとえば，引数を(少なくとも)ひとつとる関数では，関数の最初に，

```
if (args->id != _CELL)
    return error(NEA);
```

というチェックを入れておけばよいでしょう．また，引数を(少なくとも)2つとる関数ならば，引数のチェックは次のようになります．

```
if (args->id != _CELL || args->cdr->id != _CELL)
    return error(NEA);
```

引数が3つでも，4つでも以下同様となります(このスタイルだと，引数を7つも8つもとる関数では，たいへんなことになるが，そんな関数は問題外)．

また，fsubr型のLisp関数を作る時には，Cの関数にもうひとつenvというやはりCELLP型の引数をつけます．fsubr型の関数には評価されない引数が渡されますが，往々にしてその内のいくつかはCの関数の中で評価する必要があるため，その時に使う環境を引数リストとともに渡してやるために使います．したがって，中で評価をおこなわないfsubr関数ならばenvを置く必要はありません．*apply*は，受け取られるかどうかは，まったく気にせず，fsubr型の関数を呼ぶ時には常に引数リストと環境リストを渡します．

さて，引数の数が足りていたら，処理にかかります．処理に入る前に，すべての引数をローカル変数に取り出しておく手もありますが，あまりローカル変数を多く使用するとスタック領域があっというまに吹き飛んでしまいます．なるべく，必要になったらそのつど取り出すようにしましょう．

引数を取り出したら，使う前にかならずタイプチェックをするとよいでしょう．たとえば，アトムでなければならない第1引数をローカル変数cpに取り出す場合は，

```
if ((cp = args->car)->id != _ATOM)
    return error(IAA);
```

といった具合です．

5.8.2 リスト処理の慣用句

リスト処理をする関数を記述していると，似たようなプログラムをあちこちで書いているのに気がつきます．その共通点を取り出してみたのが次の2つのパターンです．関数を自作する場合の参考にしてください．

●パターン1 ―リストの各要素を順番に処理する

```
CELLP list;
while (list->id == _CELL) {        /* リストの終わりでなければ続ける */
(リストの要素 list->car を処理する)
    list = list->cdr;              /* ポインタを進める */
}
```

あるいは,

```
for ( …… ; list->id == _CELL ; list = list->cdr) {
  (リストの要素 list->car を処理する)
}
```

リストの最後のセルの cdr は普通 nil ですが, まれに普通のアトムになっていることがあります. そのため, list がまだリストの終わりにきていないかどうかは,

```
list == nil
```

ではなく上の while 文や for 文のように

```
list->id == _CELL
```

という形で判断しなければならないわけです.

●パターン2 ―リストを作る

```
CELLP cp1, cp2 ;
(最初の要素を作る)
if (ひとつも作るべき要素がない) return (CELLP)nil ;
cp1 = cp2 = newcell() ;    ec ;        /* 最初のセルを用意する */
cp2->car = (上で作った要素)
while ( …… ) {
    (次の要素を作る)
    cp2->cdr = newcell() ;    ec ;     /* セルをつなぐ */
    cp2 = cp2->cdr ;                   /* ポインタを進める */
    cp2->car = (上で作った要素)
}
cp2->cdr = nil ;                       /* リストを閉じる */
return (cp1) ;                         /* リストの先頭へのポインタを返す */
```

リストを作る時には, 空リストになってしまう場合もあることや, 作ったリストの終わりは nil で閉じておくことなどを忘れないようにしてください.

5.8.3 関数の登録 ―inisubr.c (List 5.10)

でき上がった関数をただコンパイル, リンクしただけでは, 使えるようにはなりません. その関数の存在を Lisp インタプリタに知らしめる必要があります. ファイル inisubr.c を見てください. これが(f)subr 型関数の戸籍です.

新しい関数を登録するには，

・ini_subr()先頭での関数の型宣言(すべてCELLPとして宣言する)
・ini_subr()の中のdefsubr()呼び出し

の2つを書き加えなければなりません．

● ini_subr()　(9行～42行)

すべての(f)subr型関数の初期化をおこないます．具体的な処理はdefsubr()にまかせており，ここでは，defsubr()の引数として，

・Lisp関数の名前の文字列
・対応するCの関数の名前(すなわち，その関数のトップアドレスへのポインタ)
・関数のタイプ(lisp.hで定義されている〝_SUBR〟または〝_FSUBR〟)

を与えるだけです．

● defsubr()　(44行～54行)

(f)subr型関数の初期化をおこないます．まず，mk_atom()を呼んで，その関数の名前を持つアトムを作り，oblistに登録します．次にそのアトム構造体のメンバftype(関数のタイプ)とfptr(関数の実体へのポインタ)に，与えられた引数をセットします．

たとえばsubr関数*rogue*を登録する場合には，このdefsubr()を次のように実行します．

```
defsubr( "rogue", rogue_f, _SUBR );
```

5.8.4 リスト処理関数の作成　—fun.c　(List 5.11)

最後に，このUNDEADバージョンに装備されているLisp関数の実際の定義を簡単に見ていきましょう．まず，fun.cファイルに含まれる基本的なリスト処理の関数から説明します．

● car_f()，cdr_f()，cons_f()，eq_f()，atom_f()　(10行～86行)

いわゆる〝Lispの基本5関数〟です．それぞれのプログラムの意味は，すぐわかると思います．cons_f()は，実際の処理を全部cons()に任せていますが，これはcons()の処理を他の部分でもよく使うので，他から呼び出しやすい形にしただけのことです．cons()の中身をそのままcons_f()の中に置いても同じことです．

● equal_f() (88行〜135行)

この関数も，対応するLisp関数の機能を知っていればプログラムの意味はすぐわかるでしょう．equal_f()のおもな処理をequal()として分離したのは，cons()同様，他の所で使えるようにするためです．

● putprop_f()，get_f()，remprop_f() (137行〜213行)

この3つは，シンボルアトムの属性値を操作する関数です．各シンボルアトム構造体は，属性リストへのポインタをメンバとして持っています．属性リストは次のような形をしています．

(<属性名 1> <属性値 1> <属性名 2> <属性値 2>……<属性名 n> <属性値 n>)

したがって，その長さはかならず偶数であることに注意してください．

引数の数とタイプチェックに大騒ぎをしていますが，処理そのものはごく素直で，属性リストの中でkeyに一致するものを探し，それに対応する属性値に対してそれぞれの処理をする，というのが基本的パターンです．

5.8.5 制御関数の作成 ― control.c (List 5.12)

control.cファイルに含まれる関数は，ほとんどがfsubr型の制御関数です．また多くが関数というよりは特殊形式という名前がふさわしい独特の処理をおこないます．

● cond_f() (7行〜30行)

Lispの基本的条件分岐の手段を与える関数です．引数はcond節というリストでなければなりません．引数として少なくともひとつcond節が与えられていないとエラーになります．cond節は次のような形式のリストです．

(<条件部> <本体 1> <本体 2> …… <本体 n>)

*cond*はまず条件部を評価し，その評価値がnil以外の値であれば，本体を順に評価して最後の本体の評価値をcond全体の値として返します．条件部の評価値がnilだった場合は次のcond節について同じ処理をします．もしすべての条件部がnilになったならば，*cond*の値もnilとします．また，本体のないcond節も許されていて，その場合は，条件部の評価値がnilでなければその値が返されます．

cond_f()は引数リストをclausesという変数で受け取っています．ローカル変数keyは条件部，bodiesは本体のリスト，resultは本体の評価値を置くために使っています．最低ひとつ引数があることを確かめた後，cond節を順番に処理するためのループに入ります．

各cond節の処理は，以下のとおりです．

（1）cond節がリストでなければエラーを出す．
（2）eval()を使って条件部を評価し，評価値をkeyに置く．
（3）keyがnilでなければ，
　(3.1)bodiesに本体リストを取り出し，
　(3.2)bodiesがセルでなければ，本体が与えられていないと判断してkeyを返す．
　(3.3)bodiesの各要素を順に評価し，評価値をresultにおく．resultには最後の本体の評価値が残る．
　(3.4)bodiesを返す．
（4）keyがnilだったら，clausesをclausesのcdrに置き換える．

ひとつでも評価値がnilにならない条件部があれば，clausesに関するwhileループはreturnにより途中で終了します．したがって，cond節がなくなってループを正常に終了してしまった場合は，すべての条件部がnilになってしまった場合なので，nilを返して終了します．

● setq_f() （32行～53行）

値の代入をおこなう関数です．基本的には，

(setq a b)

が，Cの

a = b;

にあたります．しかし，setqは，複数の代入をおこなうことができます．一般には次のようになります．

(setq ＜変数1＞ ＜値1＞ ＜変数2＞ ＜値2＞ …… ＜変数n＞ ＜値n＞)

したがって，引数の数は任意ですが，偶数個でなければなりません．処理は，引数2つずつを組にしておこなうことになります．また＜変数＞はアトムでなければならず，そのアトムは評価されません．

アトムの値は，場合によって置かれている場所が違います．アトムが局所変数として，環境リストの中に値が登録されている場合は，そちらが優先し環境リストの中の値が書き換えられます．そうでない場合は大域変数としてアトム構造体の中の値が書き換えられます．この作業は実際にはsetenv()が実行しています．

また，〝nil〟，〝t〟，〝eofread〟の3つのアトムは，特別な定数として値を変えることを禁止しています．

● setenv()　(55行〜71行)

setenv()は，環境リストの中に変数varを探し，見つかった場合は，その値をvalに変えてその環境リストを返し，なければNULLポインタを返します．

環境リストの各要素は，

（＜変数＞　.　＜値＞）

の形のドット対です．したがってsetenvはenvの各要素に対し，

(1) もしその要素がセルでないならば，エラーとする．
(2) その要素のcarがvarに一致したならば，その要素のcdrをvalに置き換えて，envを返す．

という処理を順におこない，最後まで見つからなければNULLを返すようにしています．

● quote_f()　(73行〜81行)

与えられた引数(評価されていないもの)をそのまま返す関数です．したがってひとつ引数があることを確認したらそれを返せばおしまいです．

● de_f()　(83行〜97行)

expr関数定義用の関数です．与えられた引数リストは次の形をしています．

（＜関数名＞ ＜lambda list＞ ＜本体1＞ …… ＜本体n＞）

このうち，かならずなければならないのは前の2つです．ローカル変数のうち，funcは関数名を置くのに使っています．上に示した引数リストは，＜関数名＞の部分を〝lambda〟に置き換えればそのままlambda式になります．そうやって作ったlambda式へのポインタを関数名アトムの構造体中のfptrにセットし，関数タイプftypeに定数_EXPRをセットして，funcを返します．

● oblist_f()　(99行〜122行)

読み込みの際に使っている配列oblist[]から順番に要素を取り出し，それらを要素とするリストを作って返す関数です．

oblist()は引数は取りません．リストの要素となるS式は配列oblist[]から取り出します．oblist []はリストへのポインタの配列で，一種の二次元データ構造になっています．oblist()はそのリストの要素を取り出して，それらを要素とする1本のリストを作ります．

● quit_f()　(124行〜127行)

メッセージを出力して，OSに復帰する関数です．本体のquit()はmain.cファイルの中にあります．

5.8.6 数値演算関数の作成　— calc.c　(List 5.13)

このUNDEADバージョンにおける数値を扱う関数は，*plus*，*minus*の2つです．これらの共通事項は，整数でも実数でも構わないがとにかく数値を引数にとることです．この時，整数でも実数でもよいという条件のあいまいさが問題になります．そこで，このような数値関数は引数にひとつでも実数型があれば，値を実数型で返すことにします．たとえば，

```
% (plus 2 3 4)
9

% (plus 2 3.0 4)
9.000000
```

のような動作が期待されるわけですが，整数型しか出てこない計算で初めから全部浮動小数点型小数点型で計算してしまうのもむだですから，できるだけ整数型で計算し実数型が必要になった時に初めて実数型を使うようにしましょう．このような考えの下で数値演算関数用の下位の関数を3つ用意しています．

● get1arg()　(10行〜30行)

引数リストと数値アトムへのポインタを引数にとります．ポインタで示される数値アトムに引数リストの中のひとつの数値アトムをコピーし，同時にその引数がちゃんと数値アトムになっているかどうかをチェックします．引数リストのcdr，つまり第2要素から後を返し，次に備えておきます．

● setfirst()　(32行〜48行)

get1arg()がすでに存在する数値アトムへの値のコピーだったのに対し，この関数は新しく数値アトムを自由領域から用意し，そこに引数のコピーをします．*plus*などが関数値として返す数値アトムはこの関数で用意されます．ある数値アトムへのポインタにその新しい数値アトムをセットするのが目的なので，そのポインタへのポインタを引数にとります．

● toflt() (50 行〜57 行)

数値アトムへのポインタを引数にとり，そのポインタによって示される数値アトムを書き換えて浮動小数点型に変換します．

● minus_f() (59 行〜77 行)

引数をひとつだけとる数値演算関数の例です．引数の数値アトムの符号を逆転した数値アトムを返します．

● plus_f() (79 行〜106 行)

plus の関数定義です．まず，setfirst で引数リストの最初の数値を新しく自由領域から切り出した数値アトムにコピーし，この数値アトムの中身を変更していく形で，引数リストが空になるまで，つまり

args->id != _CELL

になるまで計算をすすめます．

5.8.7 入出力関数の作成 — iofunc.c (List 5.14)

入出力とはいっても，この UNDEAD バージョンではコンソールとの間の入出力だけです．本格的な入出力関数の整備は CALFO バージョンのところでおこないます．

● read_f() (9 行〜15 行)

S 式の入力をおこなう関数ですので，read_s() を呼ぶだけです．

● print_f()，prin1_f()，princ_f() (17 行〜41 行)

ここにある 3 つの出力関数 *print*，*prin1*，*princ* の機能の違いは Table 5.1 のようになります．これらの関数の間にはわずかな違いしかないので，それらをひとつにまとめたのが，プログラム中の関数 prin() です．上位にある *print*，*prin1*，*princ* はこの prin() に，特殊文字に配慮するかどうかを示すフラグを渡すだけです．

	特殊文字に対する配慮	改行コード
print	○	○
prin1	○	×
princ	×	×

Table 5.1 3 つの出力関数の違い

prin1 の出力モードと *princ* の出力モードの違いについては，「5.3 S式の表示」を参照してください．また，*print* は *prin1*＋*terpri* と等しいので，*prin1* と同様の処理をおこなった後で改行コードを出力します．

● prin()　(43行～53行)

まず，引数のチェックをします．引数がない時は〝Not enough arguments〟エラーになります．次に，上位の関数が princ でなければ，特殊文字に対する配慮をつけて渡されたS式を表示します．最後に，表示した引数そのものを関数の戻り値として返します．

● terpri_f()　(55行～59行)

改行のみをおこなう関数です．改行コードを出力して，nil を返します．

List 5.1　lisp.h

```
 1: /*                                  */
 2: /*  Lisp Interpreter Header File   */
 3: /*                                  */
 4:
 5: #include    <stdio.h>
 6:
 7: /*                                  */
 8: /*  type and structure definition  */
 9: /*                                  */
10:
11: typedef unsigned char   uchar;
12: typedef uchar           *STR;
13:
14: typedef struct  cell {
15:     char    id;
16:     struct  cell    *car;
17:     struct  cell    *cdr;
18: } CELL;
19:
20: typedef CELL    *CELLP;
21:
22: typedef struct  atom {
23:     char    id;
24:     CELLP   value;
25:     CELLP   plist;
26:     STR     name;
27:     char    ftype;
28:     CELLP   fptr;
29: } ATOM;
30:
31: typedef ATOM    *ATOMP;
32:
33: typedef struct num {
34:     char    id;
35:     union body {
36:         struct num *ptr;
37:         long    fix;
38:         double  flt;
39:     } value:
40: } NUM;
41:
42: typedef NUM     *NUMP;
```

```
 43:
 44: /*                              */
 45: /*   tag number and mask of     */
 46: /*       id and function type   */
 47: /*                              */
 48:
 49: #define _CELL     1 ⎫
 50: #define _ATOM     2 ⎬ 構造体タグ
 51: #define _FIX      3 ⎪
 52: #define _FLT      4 ⎭
 53:
 54: #define _NFUNC   0x01      /* 0001 */ ⎫
 55: #define _SUBR    0x06      /* 0110 */ ⎬ 関数ボディタグ
 56: #define _EXPR    0x04      /* 0100 */ ⎪
 57: #define _FSUBR   0x02      /* 0010 */ ⎭
 58:
 59: #define _UD      0x01      /* 0001 */ ⎫
 60: #define _SR      0x02      /* 0010 */ ⎬ 関数タイプのマスク
 61: #define _EA      0x04      /* 0100 */ ⎭
 62:
 63: /*              */
 64: /*  size of ... */
 65: /*              */
 66:
 67: #define CELLSIZ      0x1996       /* 6550 (Max of large model) */
 68: #define ATOMSIZ      0x0500       /* 1280 */
 69: #define STRSIZ       0x1000       /* 4096 */
 70: #define NUMSIZ       0x1000       /* 4096 */
 71: #define TABLESIZ     64
 72: #define LINESIZ      100
 73: #define NAMLEN       100
 74:
 75: /*             */
 76: /*  switches   */
 77: /*             */
 78:
 79: #define ESCON    1
 80: #define ESCOFF   0
 81: #define ON       1
 82: #define OFF      0
 83:
 84: #define TOP 0
 85: #define UNDER    1
 86:
 87: #define TRUE     (-1)
 88: #define FALSE    0
 89:
 90: #define NONERR   0
 91: #define ERR      1
 92: #define ERROK    (-1)
 93:
 94: /*               */
 95: /*  error check  */
 96: /*               */
 97:
 98: #define     ec   if(err)return(NULL)
 99:
100: /*                */
101: /*  error number  */
102: /*                */
103:
104: #define STRUP    1   /* String area used up */
105: #define NUMUP    2   /* Number area used up */
106: #define ATOMUP   3   /* Atom area used up */
107: #define CELLUP   4   /* Cell area used up */
108: #define ULO      5   /* Unidentified Lisp Object */
109: #define PSEXP    6   /* Pseudo S-expression */
110: #define CTRLIN   7   /* Controll CHR. in the text */
```

```
111: #define UDF      8    /* Undefined function */
112: #define IFF      9    /* Illegal function form */
113: #define NEA      10   /* Not enough arguments */
114: #define IAA      11   /* Illegal argument--Atom required */
115: #define IAN      12   /* Illegal argument--Number required */
116: #define IAL      13   /* Illegal argument--List required */
117: #define IAAL     14   /* Illegal arg.--Atom or List required */
118: #define IAAN     15   /* Illegal arg.--Atom or Number required */
119: #define IALN     16   /* Illegal arg.--List or Number required */
120: #define IAF      17   /* Illegal argument--Fix Num required */
121: #define IAFL     18   /* Illegal argument--Float Num required */
122: #define ILS      19   /* Illegal structure (Illegal form) */
123: #define IASSL    20   /* Illegal A-list */
124: #define IPL      21   /* Illegal property list */
125: #define EIA      22   /* Environment is an Atom */
126: #define EHA      23   /* Environment has an Atom */
127: #define CCL      24   /* Condition Clause must be a List */
128: #define EOFERR   25   /* Unexpected EOF */
129: #define CCC      26   /* Cannot Change Constant value */
130: #define UNDEF    27   /* Error undefined */
131:
132: /*                     */
133: /*  external variable  */
134: /*                     */
135:
136: #ifdef  MAIN
137: #include    "defvar.h"
138: #else
139: #include    "var.h"
140: #endif
```

List 5.2 defvar.h

```
 1: /*                              */
 2: /*  define external variables   */
 3: /*                              */
 4:
 5: FILE    *cur_fpi, *cur_fpo;
 6:
 7: ATOMP   t, nil, lambda, eofread, prompt;
 8:
 9: CELLP   oblist[ TABLESIZ ];
10:
11: CELLP   celltop, freecell;
12: ATOMP   atomtop, freeatom;
13: NUMP    numtop, freenum;
14: STR     strtop, newstr;
15:
16: uchar   oneline[ LINESIZ ];
17: STR     txtp;
18:
19: int     err, err_no;
```

List 5.3 var.h

```
 1: /*                                  */
 2: /*   list of external variables  */
 3: /*                                  */
 4:
 5: extern FILE      *cur_fpi, *cur_fpo;
 6:
 7: extern ATOMP     t, nil, lambda, eofread, prompt;
 8:
 9: extern CELLP     oblist[ TABLESIZ ];
10:
11: extern CELLP     celltop, freecell;
12: extern ATOMP     atomtop, freeatom;
13: extern NUMP      numtop, freenum;
14: extern STR       strtop, newstr;
15:
16: extern uchar     oneline[ LINESIZ ];
17: extern STR       txtp;
18:
19: extern int       err, err_no;
```

List 5.4 read.c

```
 1: /*                              */
 2: /*          READ                */
 3: /*                              */
 4:
 5: #include     "lisp.h"
 6: #include     <ctype.h>
 7: #define      forever      for(;;)
 8:
 9: /*                              */
10: /*      read string             */
11: /*                              */
12:
13: static STR getstr()………プロンプトの出力及び入力文字列の取り込み
14: {
15:     if (isatty(fileno(cur_fpi)))
16:         print_s((CELLP)prompt, ESCOFF);
17:     *(txtp = oneline) = '¥0';
18:     return fgets(oneline, LINESIZ, cur_fpi);
19: }
20:
21: static skipspace()………空白文字を読みとばす
22: {
23:     STR getstr();
24:
25:     forever {
26:         while (isspace(*txtp))  ++txtp;
27:         if (*txtp != '¥0' && *txtp != ';')  return TRUE;
28:         if (getstr() == NULL)               return NULL;
29:         ec;
30:     }
31: }
32:
33: /*                              */
34: /*  read a S-expressiopn        */
35: /*                              */
36:
37: CELLP read_s(level)………S式の入力
38: int level;
39: {
40:     CELLP   escopt(), error();
41:     ATOMP   ret_atom();
```

```
 42:      NUMP     mk_num();
 43:
 44:      if (skipspace() == NULL)    return (CELLP)eofread;
 45:      else if (num(txtp))         return (CELLP)mk_num();
 46:      else if (isesc(*txtp))      return escopt(level);
 47:      else if (isprkana(*txtp))   return (CELLP)ret_atom(ON);
 48:      else if (iskanji(*txtp))    return (CELLP)ret_atom(ON);
 49:      else                        return error(CTRLIN);
 50: }
 51:
 52: static CELLP escopt(level)………特殊文字に対する処理の割り振り
 53: int level;
 54: {
 55:      CELLP    mk_list(), error();
 56:      ATOMP    ret_atom();
 57:
 58:      switch(*txtp) {
 59:          case '(':
 60:          case '[':   return mk_list(level);
 61:          case '|':
 62:          case '¥¥':  return (CELLP)ret_atom(ON);
 63:          default:    return error(PSEXP);
 64:      }
 65: }
 66:
 67: /*                    */
 68: /*  make numerical atom */
 69: /*                    */
 70:
 71: static NUMP mk_num()………数値アトムの作成
 72: {
 73:      char     type = _FIX;
 74:      uchar    numbuf[128], *bufp = numbuf;
 75:      double   atof();
 76:      long     atol();
 77:      NUMP     np, newnum();
 78:      CELLP    error();
 79:
 80:      if (*txtp == '+' || *txtp == '-')
 81:          *bufp++ = *txtp++;
 82:      while (isdigit(*txtp))
 83:          *bufp++ = *txtp++;
 84:      if (*txtp == '.') {
 85:          type = _FLT;
 86:          *bufp++ = *txtp++;
 87:      }
 88:      while (isdigit(*txtp))
 89:          *bufp++ = *txtp++;
 90:      if (tolower(*txtp) == 'e') {
 91:          type = _FLT;
 92:          *bufp++ = *txtp++;
 93:          if (*txtp == '+' || *txtp == '-')
 94:              *bufp++ = *txtp++;
 95:          while (isdigit(*txtp))
 96:              *bufp++ = *txtp++;
 97:      }
 98:      *bufp = '¥0';
 99:      np = newnum(); ec;
100:      if (type == _FIX)
101:          np->value.fix = atol(numbuf);
102:      else {
103:          np->id = _FLT;
104:          np->value.flt = atof(numbuf);
105:      }
106:      return np;
107: }
108:
109: /*                                          */
```

```
110: /*  If Symbol is not defined, make atom */
111: /*  Otherwise search OBLIST             */
112: /*                                      */
113:
114: static ATOMP ret_atom()………入力文字列で与えられる印字名のシンボルを返す
115: {
116:     uchar    nambuf[ NAMLEN+1 ];
117:     ATOMP    ap, old_atom(), mk_atom();
118:
119:     getname(nambuf);
120:     if ((ap = old_atom(nambuf)) == NULL)
121:         return mk_atom(nambuf);
122:     return ap;
123: }
124:
125: static hash(nam)………ハッシュキーの計算
126: STR nam;
127: {
128:     unsigned int     i = 0;
129:
130:     while (*nam != '¥0') i += *nam++;
131:     return (i % TABLESIZ);
132: }
133:
134: ATOMP old_atom(nam)………その印字名が oblist に登録されているか？
135: STR nam;
136: {
137:     int      i = 0;
138:     ATOMP    ap;
139:     CELLP    cp;
140:
141:     i = hash(nam);
142:     for (cp = oblist[i]; cp != (CELLP)nil; cp = cp->cdr) {
143:         ap = (ATOMP)cp->car;
144:         if (strcmp(ap->name, nam) == 0)     return ap;
145:     }
146:     return NULL;
147: }
148:
149: ATOMP mk_atom(nam)………シンボルを作り，oblist に登録する
150: STR nam;
151: {
152:     ATOMP    ap, mk_sub();
153:
154:     ap = mk_sub(nam); ec;
155:     intern(ap);
156:     return ap;
157: }
158:
159: static ATOMP mk_sub(nam)………シンボルの構造体に値をセットする
160: STR nam;
161: {
162:     int      length = strlen(nam) + 1;
163:     ATOMP    ap, newatom();
164:     STR      strcpy();
165:     CELLP    error();
166:
167:     if (newstr + length > strtop + STRSIZ)
168:         return (ATOMP)error(STRUP);
169:     else {
170:         nam = strcpy(newstr, nam);
171:         newstr += length;
172:     }
173:     ap = newatom(); ec;
174:     ap->value = (CELLP)ap;
175:     ap->name = nam;
176:     ap->plist = (CELLP)nil;
```

```
177:     ap->ftype = _NFUNC;
178:     return ap;
179: }
180:
181: intern(ap)………シンボルを oblist に登録する
182: ATOMP ap;
183: {
184:     int      i = 0;
185:     CELLP    cp, newcell();
186:
187:     cp = newcell(); ec;
188:     i = hash(ap->name);
189:     cp->car = (CELLP)ap;
190:     cp->cdr = oblist[i];
191:     oblist[i] = cp;
192: }
193:
194: static getname(strp)………シンボルの印字名をとり出す
195: STR strp;
196: {
197:     int      i, ifesc = OFF;
198:     CELLP    error();
199:     STR      getstr();
200:
201:     for (i = 1; i<NAMLEN; ++i) {
202:         while (*txtp == '|') {
203:             if (ifesc)  ifesc = OFF;
204:             else        ifesc = ON;
205:             ++txtp;
206:         }
207:         if (!ifesc) {                                              ┐
208:             if (*txtp == '¥¥')  ++txtp;                            │
209:             else if (isspace(*txtp) || isesc(*txtp)) {             │
210:                 if (*txtp == ';')   *txtp = '¥0';                  ├ マルチプルエスケープでないとき
211:                 *strp = '¥0';                                      │
212:                 return;                                            │
213:             }                                                      ┘
214:         }
215:         if (*txtp == '¥n' || *txtp == '¥0') {                      ┐
216:             if (getstr() == NULL)                                  │
217:                 return (int)error(EOFERR);                         ├ 新しい入力文字列のとり込み
218:             --i;                                                   │
219:             continue;                                              ┘
220:         }
221:         if (iskanji(*txtp)) {                                      ┐
222:             *strp++ = *txtp++;                                     │
223:             if (*txtp == '¥0' && getstr() == NULL) {               │
224:                 *strp = '¥0';                                      │
225:                 return;                                            │
226:             }                                                      ├ 漢字コードのとり込み
227:             else if(iskanji2(*txtp)) {                             │
228:                 *strp++ = *txtp++;                                 │
229:                 ++i;                                               │
230:                 continue;                                          │
231:             }                                                      ┘
232:         }
233:         if (!isprkana(*txtp))
234:             return (int)error(CTRLIN);
235:         *strp++ = *txtp++;
236:     }
237:     *strp = '¥0';
238: }
239:
240: /*                                  */
241: /*  make list from S expression     */
242: /*                                  */
243:
244: #define     SUP         0
```

```
245: #define     NORM        1
246:
247: static CELLP mk_list(level)………リストの作成
248: int level;
249: {
250:     char    mode;
251:     CELLP   cp1, cp2;
252:     CELLP   cp;
253:     CELLP   newcell(), error();
254:
255:     if (*txtp++ == '[') mode = SUP;
256:     else                mode = NORM;
257:
258:     if (skipspace() == NULL)    return error(EOFERR);
259:     if (*txtp == ')') {
260:         ++txtp;
261:         return (CELLP)nil;
262:     }                                                    (), []はnilになる
263:     if (*txtp == ']') {
264:         if (mode == SUP || level == TOP)
265:             ++txtp;
266:         return (CELLP)nil;
267:     }
268:     if (*txtp == '.')           return error(PSEXP);
269:     cp1 = cp = newcell(); ec;
270:     getcar(cp1, UNDER); ec;
271:     if (skipspace() == NULL)    return error(EOFERR);
272:     while (*txtp != ')' && *txtp != ']') {
273:         if (*txtp == '.') {
274:             ++txtp;
275:             if (skipspace() == NULL)          return error(EOFERR);  ドット対を
276:             if (*txtp == ')' || *txtp == ']')   return error(PSEXP);   つくる
277:             getcdr(cp1, UNDER); ec;
278:             break;
279:         }
280:         cp2 = newcell(); ec;
281:         cp1->cdr = cp2;
282:         getcar(cp2, UNDER); ec;
283:         cp1 = cp2;
284:         if (skipspace() == NULL)    return error(EOFERR);
285:     }
286:     if (*txtp == ']')
287:         if (mode == NORM && level == UNDER) return cp;
288:     ++txtp;
289:     return cp;
290: }
291:
292: static getcar(cp, level)
293: CELLP cp;
294: int level;
295: {
296:     CELLP   cp1, read_s();
297:
298:     cp1 = read_s(level); ec;
299:     cp->car = cp1;
300: }
301:
302: static getcdr(cp, level)
303: CELLP cp;
304: int level;
305: {
306:     CELLP   cp1, read_s(), error();
307:
308:     cp1 = read_s(level); ec;
309:     cp->cdr = cp1;
310:     if (skipspace() == NULL)            return (int)error(EOFERR);
311:     if (*txtp != ')' && *txtp != ']')   return (int)error(PSEXP);
312: }
```

```
313:
314: int num(x)………文字列が数値として読み込めるか？
315: STR x;
316: {
317:     if (isdigit(*x))    return TRUE;
318:     if (*x == '-' || *x == '+')
319:         if (isdigit(*++x))  return TRUE;
320:     return FALSE;
321: }
322:
323: int isesc(c)………特列文字かどうか？
324: uchar c;
325: {
326:     switch (c) {
327:         case ' ':
328:         case '#':
329:         case '.':
330:         case ';':
331:         case '(':
332:         case ')':
333:         case '[':
334:         case ']':
335:         case '¥¥':
336:         case '^':
337:         case ',':
338:         case '|':
339:         case '¥'':  return TRUE;
340:         default:    return FALSE;
341:     }
342: }
343:
344: int isprkana(c)
345: uchar c;
346: {
347:     if ((c >= ' ' ) && (c <= '~') ) return TRUE;
348:     if ((c >= 0xa1) && (c <= 0xdf)) return TRUE;
349:     return FALSE;
350: }
351:
352: int iskanji(c)
353: uchar c;
354: {
355:     if ((c > 0x80) && (c < 0xa0))   return TRUE;
356:     if ((c > 0xdf) && (c < 0xf1))   return TRUE;
357:     return FALSE;
358: }
359:
360: int iskanji2(c)
361: uchar c;
362: {
363:     if ((c > 0x3f) && (c < 0x7f))   return TRUE;
364:     if ((c > 0x7f) && (c < 0xfd))   return TRUE;
365:     return FALSE;
366: }
```

（343〜366行）MS-Cには概に定義されているので MS-Cを使用する人には必要ない

List 5.5 print.c

```
1: /*                                  */
2: /*          PRINT                   */
3: /*                                  */
4:
5: #include    "lisp.h"
6: #include    <ctype.h>
7: #define     forever     for(;;)
```

```
 8:
 9: print_s(cp, mode) ………S式の表示
10: CELLP cp;
11: int mode;
12: {
13:     if (cp->id != _CELL)
14:         pri_atom(cp, mode);
15:     else {
16:         fputc('(', cur_fpo);
17:         forever {
18:             print_s(cp->car, mode);
19:             cp = cp->cdr;
20:             if (cp->id != _CELL) break;
21:             fputc(' ', cur_fpo);
22:         }
23:         if (cp != (CELLP)nil) {
24:             fprintf(cur_fpo, " . ");
25:             pri_atom(cp, mode);
26:         }
27:         fputc(')', cur_fpo);
28:     }
29: }
30:
31: pri_atom(cp, mode) ………アトムの表示
32: CELLP cp;
33: int mode;
34: {
35:     switch (cp->id) {
36:         case _FIX:  fprintf(cur_fpo, "%ld", ((NUMP)cp)->value.fix);
37:                     break;
38:         case _FLT:  fprintf(cur_fpo, "%#.6g", ((NUMP)cp)->value.flt);
39:                     break;
40:         case _ATOM: putstr(mode, ((ATOMP)cp)->name);
41:                     break;
42:         default:    error(ULO);
43:     }
44: }
45:
46: static putstr(mode, tp) ………シンボルの印字名を表示
47: int mode;
48: STR tp;
49: {
50:     if (mode == ESCOFF)
51:         fprintf(cur_fpo, "%s", tp);
52:     else if (*tp == '¥0')
53:         fprintf(cur_fpo, "||");
54:     else {
55:         if (num(tp))
56:             fputc('¥¥', cur_fpo);
57:         do {
58:             if (iskanji(*tp) && iskanji2(*(tp+1))) {
59:                 fputc(*tp++, cur_fpo);
60:                 fputc(*tp++, cur_fpo);
61:             }
62:             else if (!isprkana(*tp)) {
63:                 fprintf(cur_fpo, "#¥¥%03d", *tp++);
64:             }
65:             else {
66:                 if (isesc(*tp))
67:                     fputc('¥¥', cur_fpo);
68:                 fputc(*tp++, cur_fpo);
69:             }
70:         } while (*tp != '¥0');
71:     }
72: }
```

List 5.6 gbc.c

```
 1: /*                                      */
 2: /*              GBC                     */
 3: /*                                      */
 4: 
 5: #include      "lisp.h"
 6: 
 7: CELLP newcell() ………セル構造体を自由領域から取り出す
 8: {
 9:     CELLP   cp, error();
10: 
11:     if (freecell == (CELLP)nil)
12:         return error(CELLUP);
13:     cp = freecell;
14:     freecell = freecell->cdr;
15:     cp->cdr = (CELLP)nil;
16:     return cp;
17: }
18: 
19: ATOMP newatom()
20: {
21:     ATOMP   ap;
22:     CELLP   error();
23: 
24:     if (freeatom == nil)
25:         return (ATOMP)error(ATOMUP);
26:     ap = freeatom;
27:     freeatom = (ATOMP)freeatom->plist;
28:     return ap;
29: }
30: 
31: NUMP newnum()
32: {
33:     NUMP    np;
34:     CELLP   error();
35: 
36:     if (freenum == (NUMP)nil)
37:         return (NUMP)error(NUMUP);
38:     np = freenum;
39:     freenum = freenum->value.ptr;
40:     return np;
41: }
```

List 5.7 eval.c

```
 1: /*                                      */
 2: /*              EVAL and APPLY          */
 3: /*                                      */
 4: 
 5: #include      "lisp.h"
 6: 
 7: CELLP eval(form, env)………S式の評価
 8: CELLP form, env;
 9: {
10:     CELLP   cp, apply(), atomvalue(), evallist(), error();
11:     ATOMP   func;
12: 
13:     switch (form->id) {
14:         case _ATOM:
15:             cp = atomvalue((ATOMP)form, env);
16:             break;
17:         case _FIX:
18:         case _FLT:
19:             return form;
20:         case _CELL:
```

```
21:             func = (ATOMP)form->car;
22:             if (eval_arg_p(func)) {
23:                 cp = evallist(form->cdr, env);
24:                 if (err) break;
25:             }
26:             else
27:                 cp = form->cdr;
28:             cp = apply((CELLP)func, cp, env);
29:             break;
30:         default:
31:             error(ULO);
32:     }
33:     if (err == ERR) {
34:         pri_err(form);
35:         return NULL;
36:     }
37:     return cp;
38: }
39:
40: static int eval_arg_p(func)………関数の引数を評価するか否かを決定する
41: ATOMP func;
42: {
43:     if (func->id == _ATOM && func->ftype & _EA)
44:         return TRUE;
45:     if (func->id == _CELL && ((CELLP)func)->car == (CELLP)lambda)
46:         return TRUE;
47:     return FALSE;
48: }
49:
50: static CELLP evallist(args, env)………関数の引数リストを作る
51: CELLP args, env;
52: {
53:     CELLP   cp1, cp2, newcell(), eval();
54:
55:     if (args->id != _CELL)
56:         return (CELLP)nil;
57:     cp1 = cp2 = newcell(); ec;
58:     cp1->car = eval(args->car, env); ec;
59:     args = args->cdr;
60:     while (args->id == _CELL) {
61:         cp1->cdr = newcell(); ec;
62:         cp1 = cp1->cdr;
63:         cp1->car = eval(args->car, env); ec;
64:         args = args->cdr;
65:     }
66:     cp1->cdr = (CELLP)nil;
67:     return cp2;
68: }
69:
70: static CELLP atomvalue(ap, env)………シンボルの値を取り出す
71: ATOMP ap;
72: CELLP env;
73: {
74:     CELLP   error();
75:
76:     while (env->id == _CELL) {
77:         if (env->car->id != _CELL)
78:             return error(EHA);
79:         if (env->car->car == (CELLP)ap)
80:             return env->car->cdr;
81:         env = env->cdr;
82:     }
83:     return ap->value;
84: }
85:
86: static CELLP apply(func, args, env)………関数の評価
87: CELLP func, args, env;
88: {
```

```
 89:     CELLP   cp, (*funcp)(), bodies, result = (CELLP)nil;
 90:     CELLP   bind(), error();
 91:     char    funtype;
 92:
 93:     switch (func->id) {
 94:         case _ATOM:
 95:             funtype = ((ATOMP)func)->ftype;
 96:             if (funtype & _UD)
 97:                 return error(UDF);
 98:             if (funtype & _SR) {
 99:                 funcp = (CELLP (*)())((ATOMP)func)->fptr;
100:                 if (funtype & _EA)
101:                     return (*funcp)(args);
102:                 else
103:                     return (*funcp)(args, env);
104:             }
105:             func = ((ATOMP)func)->fptr;
106:         case _CELL:
107:             if (func->cdr->id != _CELL)
108:                 return error(IFF);
109:             if (func->car == (CELLP)lambda) {………lambda式の評価
110:                 bodies = func->cdr->cdr;
111:                 cp = bind(func->cdr->car, args, env); ec;
112:                 for (; bodies->id == _CELL; bodies = bodies->cdr) {  ⎱ Implicitな
113:                     result = eval(bodies->car, cp); ec;               ⎰ prognの処理
114:                 }
115:                 return result;
116:             }
117:         default:
118:             return error(IFF);
119:     }
120: }
121:
122: CELLP bind(keys, values, env)………環境に対し変数の束縛を行なう
123: CELLP keys, values, env;
124: {
125:     CELLP   push(), error();
126:
127:     if (keys != (CELLP)nil && keys->id == _ATOM) {
128:         env = push(keys, values, env); ec;
129:         return env;
130:     }
131:     while (keys->id == _CELL) {
132:         if (values->id != _CELL)
133:             return error(NEA);
134:         env = push(keys->car, values->car, env); ec;
135:         keys = keys->cdr;
136:         values = values->cdr;
137:     }
138:     if (keys != (CELLP)nil && keys->id == _ATOM) {
139:         env = push(keys, values, env); ec;
140:     }
141:     return env;
142: }
143:
144: static CELLP push(key, value, env)………環境リストに新しい項をつけ加える
145: CELLP key, value, env;
146: {
147:     CELLP   cp, newcell();
148:
149:     cp = newcell(); ec;
150:     cp->cdr = env;
151:     env = cp;
152:     env->car = newcell(); ec;
153:     env->car->car = key;
154:     env->car->cdr = value;
155:     return cp;
156: }
```

List 5.8 error.c

```
 1: /*                             */
 2: /*          ERROR              */
 3: /*                             */
 4: /*  define error massage       */
 5: /*  set error code             */
 6: /*      if error happened      */
 7: /*                             */
 8:
 9: #include    "lisp.h"
10:
11: static STR err_msg[] = {
12:     "I don't know what happend",
13:     "String area used up",
14:     "Number area used up",
15:     "Atom area used up",
16:     "Cell area used up",
17:     "Unidentified Lisp Object",
18:     "Pseudo S expression",
19:     "Ctrl character sneaks in",
20:     "Undefined function",
21:     "Illegal function form",
22:     "Not enough arguments",
23:     "Illegal argument--Atom required",
24:     "Illegal argument--Number required",
25:     "Illegal argument--List required",
26:     "Illegal arg.--Atom or List required",
27:     "Illegal arg.--Atom or Number required",
28:     "Illegal arg.--List or Number required",
29:     "Illegal argument--Fix Num required",
30:     "Illegal argument--Float Num required",
31:     "Illegal structure",
32:     "Illegal association list",
33:     "Illegal property list",
34:     "Environment is an atom",
35:     "Environment has an atom",
36:     "Condition Clause must be a List",
37:     "Unexpected EOF",
38:     "Can't Change Constant value",
39:     "Unidentified error",
40: };
41:
42: CELLP error(code) ………エラーフラグのセット
43: int code;
44: {
45:     err = ERR;
46:     err_no = code;
47:     return NULL;
48: }
49:
50: pri_err(form) ………エラーメッセージの表示
51: CELLP form;
52: {
53:     fprintf(stderr, "¥nOops!¥n");
54:     fprintf(stderr, "Error No.%d : %s¥n", err_no, err_msg[ err_no ]);
55:     fprintf(stderr, "At ");
56:     if ((ATOMP)form == nil)
57:         fprintf(stderr, "toplevel");
58:     else {
59:         cur_fpo = stderr;
60:         print_s(form, ESCON);
61:     }
62:     fprintf(stderr, "¥n¥n");
63:     err = ERROK;
64: }
```

List 5.9 main.c

```
 1: /*                                                */
 2: /*                    MAIN                        */
 3: /*                                                */
 4: /*  Lisp Main Routine and Initialize System      */
 5: /*                                                */
 6:
 7: #define     MAIN
 8: #include    "lisp.h"
 9: #include    <signal.h>
10: #define     forever      for(;;)
11:
12: main()
13: {
14:     greeting();
15:     init();
16:     forever {
17:         if (err == ERR) pri_err((CELLP)nil); ………トップレベルで起きたエラーの処理
18:         reset_err();
19:         toplevel_f();
20:         reset_stdin();
21:     }
22: }
23:
24: toplevel_f() ………トップレベルループ
25: {
26:     CELLP   arg;
27:     CELLP   read_s(), eval(), error();
28:
29:     forever {
30:         arg = read_s(TOP);                          ┐
31:         if (arg == (CELLP)eofread)  break;          ├ READ
32:         ec;                                         ┘
33:         arg = eval(arg, (CELLP)nil);                ┐
34:         switch (err) {                              │
35:             case ERROK:                             ├ EVAL
36:             case ERR:   return (int)(err = ERROK);  │
37:         }                                           ┘
38:         print_s(arg, ESCON);                        ┐
39:         ec;                                         │
40:         if (isatty(fileno(cur_fpi)))                ├ PRINT
41:             fputc('¥n', cur_fpo);                   │
42:         fputc('¥n', cur_fpo);                       ┘
43:     }
44: }
45:
46: static reset_err() ………エラーの後処理
47: {
48:     cur_fpi = stdin;
49:     cur_fpo = stdout;
50:     err = NONERR;
51:     txtp = oneline;
52:     *txtp = '¥0';
53: }
54:
55: static init() ………システムの初期化
56: {
57:     char    *malloc();
58:     int     quit();
59:     int     i;
60:     CELLP   cp;
61:     ATOMP   ap;
62:     NUMP    np;
63:
64:     freecell = celltop = (CELLP)malloc(sizeof(CELL) * CELLSIZ);  ┐
65:     freeatom = atomtop = (ATOMP)malloc(sizeof(ATOM) * ATOMSIZ);  ├ 各領域の確保
66:     freenum = numtop = (NUMP)malloc(sizeof(NUM) * NUMSIZ);       │
67:     newstr = strtop = (STR)malloc(STRSIZ);                       ┘
```

```
 68:
 69:     if (freecell == NULL || freeatom == NULL
 70:         || freenum == NULL || newstr == NULL) {
 71:         printf("Oops! Alloc Error : Too Large Data Area.¥n");
 72:         printf("Please change --SIZ (defined in lisp.h).¥n");
 73:         exit(1);
 74:     }
 75:
 76:     nil = freeatom++;
 77:
 78:     for (cp = celltop; cp < celltop + CELLSIZ; ++cp) { ………セル領域の初期化
 79:         cp->id = _CELL;
 80:         cp->car = (CELLP)nil;
 81:         cp->cdr = cp + 1;
 82:     }
 83:     (--cp)->cdr = (CELLP)nil;
 84:
 85:     for (ap = atomtop + 1; ap < atomtop + ATOMSIZ; ++ap) { ……アトム領域の初期化
 86:         ap->id = _ATOM;
 87:         ap->plist = (CELLP)(ap + 1);
 88:     }
 89:     (--ap)->plist = (CELLP)nil;
 90:
 91:     for (np = numtop; np < numtop + NUMSIZ; ++np) { ………数値アトム領域の初期化
 92:         np->id = _FIX;
 93:         np->value.ptr = np + 1;
 94:     }
 95:     (--np)->value.ptr = (NUMP)nil;
 96:
 97:     for (i = 0; i < TABLESIZ; ++i) ………ハッシュテーブルの初期化
 98:         oblist[i] = (CELLP)nil;
 99:
100:     mk_sys_atoms();
101:     ini_subr();
102:     signal( SIGINT, quit ); ………Ctrl-C割り込みの飛先を変える
103: }
104:
105: static mk_nil() ………nilの設定
106: {
107:     char    *strcpy();
108:     char    *s = strcpy(newstr, "nil");
109:
110:     newstr += strlen(s) + 1;
111:     nil->id = _ATOM;
112:     nil->value = (CELLP)nil;
113:     nil->plist = (CELLP)nil;
114:     nil->name = s;
115:     nil->ftype = _NFUNC;
116:     nil->fptr = (CELLP)nil;
117:     intern(nil);
118: }
119:
120: static mk_sys_atoms() ………システムが必要とするシンボルを作る
121: {
122:     ATOMP   mk_atom();
123:
124:     mk_nil();
125:     t = mk_atom("t");
126:     lambda = mk_atom("lambda");
127:     eofread = mk_atom("EOF");
128:     prompt = mk_atom("% ");
129: }
130:
131: static greeting()
132: {
133:     fprintf(stdout,"¥n");
134:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
135:     fprintf(stdout,"¥t          U N D E A D¥n");
```

```
136:     fprintf(stdout,"¥t  Will o'Lisp Version 0.00¥n");
137:     fprintf(stdout,"¥t       (C)  1986, Feb¥n¥n");
138:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
139: }
140:
141: quit()
142: {
143:     fprintf(stdout, "¥n¥n¥tMay the force be with you!¥n");
144:     fprintf(stdout, "¥t¥t¥t--From Will o'Lisp with Love.¥n¥n");
145:     exit(0);
146: }
147:
148: reset_stdin() ………EOFによりconsoleが閉じてしまうのを防ぐ
149: {
150:     if (isatty(fileno(stdin)))
151:         rewind(stdin);
152:     else
153:         freopen( "CON", "r", stdin );
154: }
```

List 5.10 inisubr.c

```
 1: /*                                          */
 2: /*                  INISUBR                 */
 3: /*                                          */
 4: /*  initialize SUBR and FSUBR type function */
 5: /*                                          */
 6:
 7: #include "lisp.h"
 8:
 9: ini_subr() ………組み込み関数の登録
10: {
11:     CELLP car_f(), cdr_f(), cons_f();
12:     CELLP atom_f(), eq_f(), equal_f();
13:     CELLP quote_f(), de_f(), cond_f();
14:     CELLP setq_f(), oblist_f(), quit_f();
15:     CELLP putprop_f(), get_f(), remprop_f();
16:     CELLP read_f(), terpri_f();
17:     CELLP print_f(), prin1_f(), princ_f();
18:     CELLP minus_f(), plus_f();
19:
20:     defsubr("car",      car_f,      _SUBR);
21:     defsubr("cdr",      cdr_f,      _SUBR);
22:     defsubr("cons",     cons_f,     _SUBR);
23:     defsubr("atom",     atom_f,     _SUBR);
24:     defsubr("eq",       eq_f,       _SUBR);
25:     defsubr("equal",    equal_f,    _SUBR);
26:     defsubr("quote",    quote_f,    _FSUBR);
27:     defsubr("de",       de_f,       _FSUBR);
28:     defsubr("cond",     cond_f,     _FSUBR);
29:     defsubr("setq",     setq_f,     _FSUBR);
30:     defsubr("oblist",   oblist_f,   _SUBR);
31:     defsubr("quit",     quit_f,     _SUBR);
32:     defsubr("putprop",  putprop_f,  _SUBR);
33:     defsubr("get",      get_f,      _SUBR);
34:     defsubr("remprop",  remprop_f,  _SUBR);
35:     defsubr("read",     read_f,     _SUBR);
36:     defsubr("terpri",   terpri_f,   _SUBR);
37:     defsubr("print",    print_f,    _SUBR);
38:     defsubr("prin1",    prin1_f,    _SUBR);
39:     defsubr("princ",    princ_f,    _SUBR);
40:     defsubr("minus",    minus_f,    _SUBR);
41:     defsubr("plus",     plus_f,     _SUBR);
42: }
43:
```

```
44: static defsubr(name, funcp, type)
45: STR name;
46: CELLP (*funcp)();
47: char type;
48: {
49:     ATOMP   ap, mk_atom();
50:
51:     ap = mk_atom(name); ec;
52:     ap->ftype = type;
53:     ap->fptr = (CELLP)funcp;
54: }
```

List 5.11 fun.c

```
 1: /*                                                  */
 2: /*              FUNC                                */
 3: /*                                                  */
 4: /*          Lisp functions                          */
 5: /*                                                  */
 6:
 7: #include    "lisp.h"
 8: #define     forever     for(;;)
 9:
10: /*                              */
11: /*  pure lisp functions         */
12: /*                              */
13:
14: CELLP car_f(args)
15: CELLP args;
16: {
17:     CELLP   error();
18:
19:     if (args->id != _CELL)
20:         return error(NEA);
21:     if (args->car == (CELLP)nil)
22:         return (CELLP)nil;
23:     if (args->car->id != _CELL)
24:         return error(IAL);
25:     return args->car->car;
26: }
27:
28: CELLP cdr_f(args)
29: CELLP args;
30: {
31:     CELLP   error();
32:
33:     if (args->id != _CELL)
34:         return error(NEA);
35:     if (args->car == (CELLP)nil)
36:         return (CELLP)nil;
37:     if (args->car->id != _CELL)
38:         return error(IAL);
39:     return args->car->cdr;
40: }
41:
42: CELLP cons_f(args)
43: CELLP args;
44: {
45:     CELLP   cp, cons();
46:
47:     if (args->id != _CELL || args->cdr->id != _CELL)
48:         return error(NEA);
49:     cp = cons(args->car, args->cdr->car); ec;
50:     return cp;
51: }
```

```
 52:
 53: CELLP cons(arg1, arg2)
 54: CELLP arg1, arg2;
 55: {
 56:     CELLP   cp, newcell();
 57:
 58:     cp = newcell(); ec;
 59:     cp->car = arg1;
 60:     cp->cdr = arg2;
 61:     return cp;
 62: }
 63:
 64: CELLP eq_f(args)
 65: CELLP args;
 66: {
 67:     CELLP   error();
 68:
 69:     if (args->id != _CELL || args->cdr->id != _CELL)
 70:         return error(NEA);
 71:     if (args->car == args->cdr->car)
 72:         return (CELLP)t;
 73:     return (CELLP)nil;
 74: }
 75:
 76: CELLP atom_f(args)
 77: CELLP args;
 78: {
 79:     CELLP   error();
 80:
 81:     if (args->id != _CELL)
 82:         return error(NEA);
 83:     if (args->car->id != _CELL)
 84:         return (CELLP)t;
 85:     return (CELLP)nil;
 86: }
 87:
 88: CELLP equal_f(args)
 89: CELLP args;
 90: {
 91:     int     equal();
 92:     CELLP   error();
 93:
 94:     if (args->id != _CELL || args->cdr->id != _CELL)
 95:         return error(NEA);
 96:     while (args->cdr->id == _CELL) {
 97:         if (!equal(args->car, args->cdr->car))
 98:             return (CELLP)nil;
 99:         args = args->cdr;
100:     }
101:     return (CELLP)t;
102: }
103:
104: equal(arg1, arg2)
105: CELLP arg1, arg2;
106: {
107:     if (arg1 == arg2)
108:         return TRUE;
109:     if (arg1->id != arg2->id)
110:         return FALSE;
111:     switch (arg1->id) {
112:         case _FIX:
113:             if (((NUMP)arg1)->value.fix == ((NUMP)arg2)->value.fix)
114:                 return TRUE;
115:             else
116:                 return FALSE;
117:         case _FLT:
118:             if (((NUMP)arg1)->value.flt == ((NUMP)arg2)->value.flt)
119:                 return TRUE;
```

```
120:            else
121:                return FALSE;
122:        case _ATOM:
123:            if (strcmp(((ATOMP)arg1)->name, ((ATOMP)arg2)->name))
124:                return FALSE;
125:            else
126:                return TRUE;
127:        case _CELL:
128:            if (equal(arg1->car, arg2->car) && equal(arg1->cdr, arg2->cdr))
129:                return TRUE;
130:            else
131:                return FALSE;
132:        default:
133:            return (int)error(ULO);
134:    }
135: }
136:
137: /*                 */
138: /*  property list  */
139: /*                 */
140:
141:
142: CELLP putprop_f(args)
143: CELLP args;
144: {
145:     CELLP   val, cp, error();
146:     ATOMP   key, ap;
147:
148:     if (args->id != _CELL
149:         || args->cdr->id != _CELL
150:         || args->cdr->cdr->id != _CELL)
151:         return error(NEA);
152:     if ((ap = (ATOMP)args->car)->id != _ATOM
153:         || (key = (ATOMP)args->cdr->cdr->car)->id != _ATOM)
154:         return error(IAA);
155:     val = args->cdr->car;
156:     cp = ap->plist;
157:     for (cp = ap->plist; cp->id == _CELL; cp = cp->cdr->cdr) {
158:         if ((ATOMP)cp->car == key)
159:             return (cp->cdr->car = val);
160:     }
161:     cp = newcell(); ec;
162:     cp->car = (CELLP)key;
163:     cp->cdr = newcell(); ec;
164:     cp->cdr->car = val;
165:     cp->cdr->cdr = ap->plist;
166:     ap->plist = cp;
167:     return val;
168: }
169:
170: CELLP get_f(args)
171: CELLP args;
172: {
173:     CELLP   cp, error();
174:     ATOMP   key, ap;
175:
176:     if (args->id != _CELL || args->cdr->id != _CELL)
177:         return error(NEA);
178:     if ((ap = (ATOMP)args->car)->id != _ATOM
179:         || (key = (ATOMP)args->cdr->car)->id != _ATOM)
180:         return error(IAA);
181:     for (cp = ap->plist; cp->id == _CELL; cp = cp->cdr->cdr) {
182:         if ((ATOMP)cp->car == key)
183:             return cp->cdr->car;
184:     }
185:     return (CELLP)nil;
186: }
187:
```

```
188: CELLP remprop_f(args)
189: CELLP args;
190: {
191:     CELLP   val, cp, error();
192:     ATOMP   key, ap;
193:
194:     if (args->id != _CELL || args->cdr->id != _CELL)
195:         return error(NEA);
196:     if ((ap = (ATOMP)args->car)->id != _ATOM
197:         || (key = (ATOMP)args->cdr->car)->id != _ATOM)
198:         return error(IAA);
199:     if ((cp = ap->plist) == (CELLP)nil)
200:         return (CELLP)nil;
201:     if ((ATOMP)cp->car == key) {
202:         ap->plist = cp->cdr->cdr;
203:         return cp->cdr->car;
204:     }
205:     for (cp = cp->cdr; cp->cdr->id == _CELL; cp = cp->cdr->cdr) {
206:         if ((ATOMP)cp->cdr->car == key) {
207:             val = cp->cdr->cdr->car;
208:             cp->cdr = cp->cdr->cdr->cdr;
209:             return val;
210:         }
211:     }
212:     return (CELLP)nil;
213: }
```

List 5.12 control.c

```
 1: /*                                                        */
 2: /*                  Control functions                     */
 3: /*                                                        */
 4:
 5: #include    "lisp.h"
 6:
 7: CELLP cond_f(clauses, env)
 8: CELLP clauses, env;
 9: {
10:     CELLP   key, bodies, result, eval(), error();
11:
12:     if (clauses->id != _CELL)
13:         return error(NEA);
14:     while (clauses->id == _CELL) {
15:         if (clauses->car->id != _CELL)
16:             return error(CCL);
17:         key = eval(clauses->car->car, env); ec;
18:         if (key != (CELLP)nil) {
19:             if ((bodies = clauses->car->cdr)->id != _CELL)
20:                 return key;
21:             while (bodies->id == _CELL) {
22:                 result = eval(bodies->car, env); ec;
23:                 bodies = bodies->cdr;
24:             }
25:             return result;
26:         }
27:         clauses = clauses->cdr;
28:     }
29:     return (CELLP)nil;
30: }
31:
32: CELLP setq_f(args, env)
33: CELLP args, env;
34: {
35:     CELLP   val, result, setenv(), eval(), error();
36:     ATOMP   var;
```

```
 37:
 38:     while (args->id == _CELL) {
 39:         if (args->cdr->id != _CELL)
 40:             return error(NEA);
 41:         var = (ATOMP)args->car;
 42:         val = eval(args->cdr->car, env); ec;
 43:         if (var->id != _ATOM)
 44:             return error(IAA);
 45:         if (var == nil || var == t || var == eofread)
 46:             return error(CCC);
 47:         result = setenv(var, val, env); ec;
 48:         if (result == NULL)
 49:             var->value = val;
 50:         args = args->cdr->cdr;
 51:     }
 52:     return val;
 53: }
 54:
 55: static CELLP setenv(var, val, env)
 56: ATOMP var;
 57: CELLP val, env;
 58: {
 59:     CELLP   error();
 60:
 61:     while (env->id == _CELL) {
 62:         if (env->car->id != _CELL)
 63:             return error(EHA);
 64:         if (env->car->car == (CELLP)var) {
 65:             env->car->cdr = val;
 66:             return env;
 67:         }
 68:         env = env->cdr;
 69:     }
 70:     return NULL;
 71: }
 72:
 73: CELLP quote_f(args, env)
 74: CELLP args, env;
 75: {
 76:     CELLP error();
 77:
 78:     if (args->id != _CELL)
 79:         return error(NEA);
 80:     return args->car;
 81: }
 82:
 83: CELLP de_f(args, env)
 84: CELLP args, env;
 85: {
 86:     CELLP val, cons(), error();
 87:     ATOMP func;
 88:
 89:     if (args->id != _CELL || args->cdr->id != _CELL)
 90:         return error(NEA);
 91:     if ((func = (ATOMP)args->car)->id != _ATOM)
 92:         return error(IAA);
 93:     val = cons((CELLP)lambda, args->cdr); ec;
 94:     func->ftype = _EXPR;
 95:     func->fptr = val;
 96:     return (CELLP)func;
 97: }
 98:
 99: CELLP   oblist_f()
100: {
101:     int     i = 0;
102:     CELLP   cp, cp1, cp2, cp3, newcell();
103:
104:     cp1 = cp = newcell(); ec;
```

```
105:     for (i = 0; i<TABLESIZ; ++i)
106:         if ((cp2 = oblist[i]) != (CELLP)nil) break;
107:     for(; cp2->cdr != (CELLP)nil; cp2 = cp2->cdr) {
108:         cp1->car = cp2->car;
109:         cp3 = newcell(); ec;
110:         cp1->cdr = cp3;
111:         cp1 = cp3;
112:     }
113:     cp1->car = cp2->car;
114:     for (++i; i<TABLESIZ; ++i)
115:         for(cp2 = oblist[i]; cp2 != (CELLP)nil; cp2 = cp2->cdr) {
116:             cp3 = newcell(); ec;
117:             cp1->cdr = cp3;
118:             cp1 = cp3;
119:             cp1->car = cp2->car;
120:         }
121:     return cp;
122: }
123:
124: CELLP quit_f()
125: {
126:     quit();
127: }
```

List 5.13 calc.c

```
 1: /*                                                    */
 2: /*                   CALC                             */
 3: /*                                                    */
 4: /*     function for calculating numbers               */
 5: /*                                                    */
 6:
 7: #include     "lisp.h"
 8: #define      forever      for(;;)
 9:
10: CELLP getlarg(args, valp)
11: CELLP args;
12: NUMP valp;
13: {
14:     char     c;
15:     CELLP    error();
16:
17:     if (args->id != _CELL)
18:         return error(NEA);
19:     if ((c = args->car->id) == _FIX) {
20:         valp->id = _FIX;
21:         valp->value.fix = ((NUMP)(args->car))->value.fix;
22:         return args->cdr;
23:     }
24:     if (c == _FLT) {
25:         valp->id = _FLT;
26:         valp->value.flt  = ((NUMP)(args->car))->value.flt;
27:         return args->cdr;
28:     }
29:     return error(IAN);
30: }
31:
32: static CELLP setfirst(args, npp)
33: CELLP args;
34: NUMP *npp;
35: {
36:     CELLP    getlarg();
37:     NUM      val, *newnum();
38:
39:     args = getlarg(args, &val);  ec;
```

```
 40:     *npp = newnum();  ec;
 41:     if (val.id == _FIX)
 42:         (*npp)->value.fix = val.value.fix;
 43:     else {
 44:         (*npp)->id = _FLT;
 45:         (*npp)->value.flt = val.value.flt;
 46:     }
 47:     return args;
 48: }
 49:
 50: toflt(np)
 51: NUMP np;
 52: {
 53:     if (np->id != _FLT) {
 54:         np->id = _FLT;
 55:         np->value.flt = (double)(np->value.fix);
 56:     }
 57: }
 58:
 59: CELLP minus_f(args)
 60: CELLP args;
 61: {
 62:     char    c;
 63:     NUMP    np, newnum();
 64:
 65:     if (args->id != _CELL)
 66:         return error(NEA);
 67:     if ((c = args->car->id) != _FIX && c != _FLT)
 68:         return error(IAN);
 69:     np = newnum();  ec;
 70:     if (c == _FIX)
 71:         np->value.fix = -((NUMP)(args->car))->value.fix;
 72:     else {
 73:         np->id = _FLT;
 74:         np->value.flt = -((NUMP)(args->car))->value.flt;
 75:     }
 76:     return (CELLP)np;
 77: }
 78:
 79: CELLP plus_f(args)
 80: CELLP args;
 81: {
 82:     CELLP   setfirst(), getlarg();
 83:     NUM     val;
 84:     NUMP    np;
 85:
 86:     args = setfirst(args, &np);  ec;
 87:     if (args == (CELLP)nil)
 88:         return (CELLP)np;
 89:     args = getlarg(args, &val);  ec;
 90:     if (np->id == _FIX) {
 91:         while(val.id == _FIX) {
 92:             np->value.fix += val.value.fix;
 93:             if (args == (CELLP)nil)
 94:                 return (CELLP)np;
 95:             args = getlarg(args, &val); ec;
 96:         }
 97:         toflt(np);
 98:     }
 99:     else    toflt(&val);
100:     forever {
101:         np->value.flt += val.value.flt;
102:         if (args == (CELLP)nil) return (CELLP)np;
103:         args = getlarg(args, &val); ec;
104:         toflt(&val);
105:     }
106: }
```

List 5.14 iofunc.c

```
 1: /*                                              */
 2: /*              IOFUNC                          */
 3: /*                                              */
 4: /*      input/output functions                  */
 5: /*                                              */
 6:
 7: #include    "lisp.h"
 8:
 9: CELLP read_f(args)
10: CELLP args;
11: {
12:     CELLP   read_s();
13:
14:     return  read_s(TOP);
15: }
16:
17: CELLP print_f(args)
18: CELLP args;
19: {
20:     CELLP   prin(), result;
21:
22:     result = prin(args, ESCON); ec;
23:     fputc('¥n', cur_fpo);
24:     return result;
25: }
26:
27: CELLP prin1_f(args)
28: CELLP args;
29: {
30:     CELLP   prin();
31:
32:     return prin(args, ESCON);
33: }
34:
35: CELLP princ_f(args)
36: CELLP args;
37: {
38:     CELLP   prin();
39:
40:     return prin(args, ESCOFF);
41: }
42:
43: static CELLP prin(args, mode)
44: CELLP args;
45: int mode;
46: {
47:     CELLP   cp, error();
48:
49:     if (args->id != _CELL)
50:         return error(NEA);
51:     print_s(cp = args->car, mode);
52:     return cp;
53: }
54:
55: CELLP terpri_f()
56: {
57:     fputc('¥n', cur_fpo);
58:     return (CELLP)nil;
59: }
```

6章　機能拡張

これまで紹介してきたUNDEADバージョンは極々小さなLisp処理系としてきちんと動作はしますが，それはまさに最低限の機能しかありませんでした．致命的なのはガベージコレクタがないことです．また，UNDEADバージョンには関数がほとんどありません．何を作るにも関数不足で手詰まりになります．本章では，これらの不足を補うため，各種の機能の追加あるいは関数の作成をおこなっていきます．

6.1 機能拡張(1) ガベージコレクタの作成
―MADIバージョン―

機能拡張の第1弾として，まずおこなうべき作業は，ガベージコレクタの作成です．この拡張によって，Will o'Lisp は実用化への第1歩を踏み出します．

6.1.1 ガベージコレクタの必要性

先にも述べたとおり，Lispは次々とセルを食い潰していく言語なのにもかかわらず，これまで述べてきたUNDEADバージョンの処理系は，それを補う術を知りません．たとえば次のような関数を定義してみましょう．

```
% (de append2 (x y)
%     (cond ((null x) y)
%           (t (cons (car x) (append2 (cdr x) y)))))
append2

% (de reverse (l)
%     (cond ((null l) nil)
%           (t (append2 (reverse (cdr l)) (cons (car l) nil)))))
reverse

% (append2 (quote (wo hu ro)) (quote (ba ma co)))
(wo hu ro ba ma co)

% (reverse (quote (wo hu ro ba ma co)))
(co ma ba ro hu wo)
```

これらの関数定義についてはここでは詳しく触れませんが，*append2* は2つのリストをつなぐ関数，*reverse* はリストの要素を逆順に並べたリストを返す関数です．さて，これらの関数がどのくらいセルを使うか考えてみますと，引数リスト，環境リスト，評価の3つに使われるセルを数えればよいのですから，オーダーのみの計算で *append2* ではリスト x の長さに比例し，*reverse* ではリスト l の長さの2乗に比例してセルを使います．長さ30のリストの反転に数千ものセルを使う計算になります．セル領域が小さいと oblist の反転すらできません．ガベージコレクタの必要性がこのことからもよくわかります．

6.1.2 Will o'Lisp のガベージコレクタ

ガベージコレクタの概念はこれまで述べたとおり，使用済みになったセル，アトムなどを再び使えるように再生することです．そのために Will o'Lisp が採る方法とは，必要とする領域がなくなった時にこのガベージコレクタを起動して一気に回収をおこなうバッチ型であることもすでに述べました．この伝統的な方法ではガベージコレクタが起動された時点で使用されているセル，アトムにマークを付け，すべての領域を検索しマークの付いていないセル，アトムを未使用のものとして自由領域に戻します．マークを付けるに当たってはその構造体の中に印を付ける方法と，構造体の中に余裕がない時は別に空き領域にビットマップを用意し，そこにビットを立てていく方法とがありますが，Will o'Lisp では前者の構造体に直接マークする方法を用います．なお，アトムという言葉は数値アトムとシンボルアトムを併せたものとして使っています．

さて，使用中か未使用かはどうやって見分けるのでしょうか．それぞれの構造体に〝お前は仕事をしているか〟と尋ねても答えるはずがありません．下っぱの構造体本人にはわからないのです．そこで，芋づる式に頭からたぐって調べねばなりません．この頭にあたるのが oblist です．oblist をたぐることで，次の物をマークすることができます．

- ・oblist 自体に使われているセル
- ・oblist に登録されているすべてのシンボルアトム
- ・そのシンボルアトムの値，属性リスト，関数ボディに使われているセル，アトム
- ・そのセルから見えるセル，数値アトム，oblist に未登録のシンボルアトム

oblist を構成するセルから始めて，car をたぐり，cdr をたぐり，oblist から見えるすべてのオブジェクトにマークするのです．最後の〝oblist に未登録のシンボルアトム〟とはまだ説明してありませんし，機能としても備っていませんが，最終段階の MAX バージョンで装備されます．

6.1.3 ソフトウェアスタック

さて，以上のマークで，すべて網羅したと思ったらそれは間違いです．ここにあるのはいわば静的な使用状況で，トップレベルでの状態でしかありません．深い関数評価の中では，これらの他にも使用中のセルが存在するのです．それは，ローカル変数の束縛を示す環境リストと下位の関数のために作られる引数リストです．これらがガベージコレクタから見えるようにしておく必要があります．

完全に作動するガベージコレクタは Lisp には必要不可欠ですが，これが，もし仮に不完全な動作をしているとすれば，こんな破壊的な作用はありません．したがって必要なものはかならずガ

ベージコレクタから見えるようにしなくてはなりません.

oblistから見えるオブジェクト以外は，大体においてCの局所変数，関数引数としてスタックに積まれています．このスタックを直接調べられればよいのですが，ここにはLispの必要とする情報以外のもの(Cの関数のリターンアドレスやベースポインタなど)もあって，直接調べるのは面倒です.

そこで，これらをガベージコレクタから見やすくするために，ソフトウェアで操作するスタックを作り，必要なものはここに積むことにします．これを以後ソフトウェアスタックと呼びます.

ソフトウェアスタックはFig. 6.1のような構造をしています．セルへのポインタの列を作り，その先頭をセルポインタへのポインタstacktopが指し，現在のスタックのある場所をセルポインタへのポインタspが示しています．したがって，

$$* \ sp$$

のような記述が，あるリストへのポインタになります.

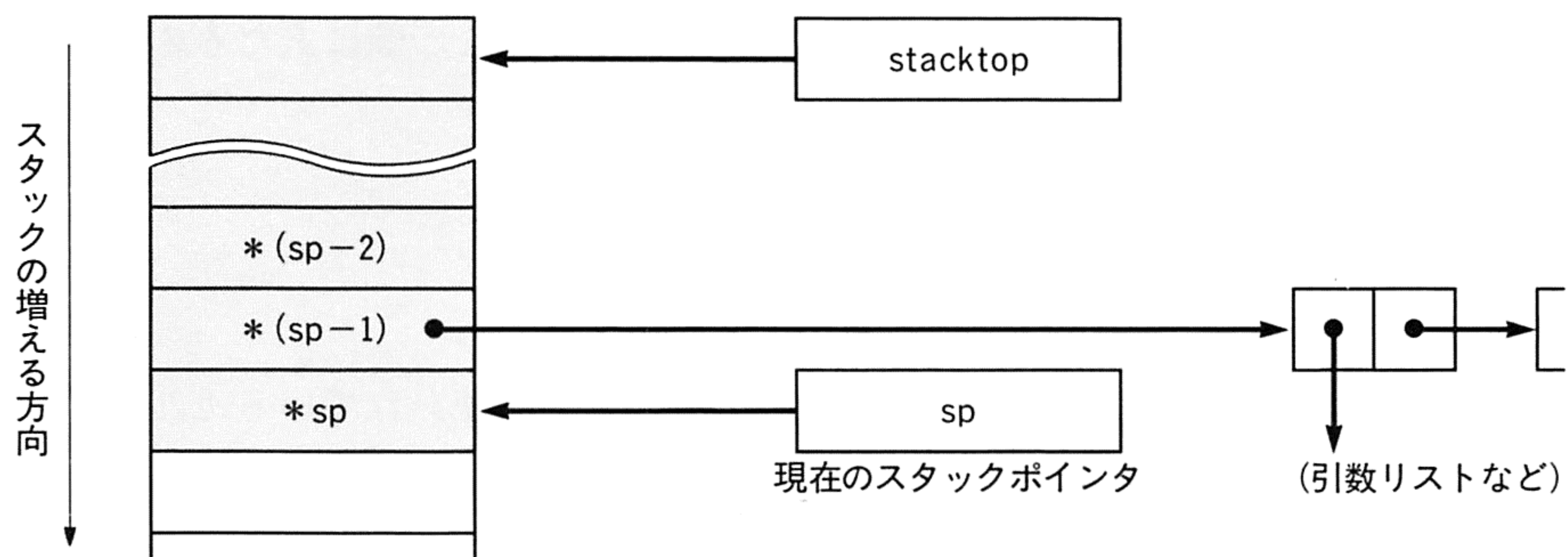



Fig. 6.1 ソフトウェアスタック

ここでソフトウェアスタック使用上の注意をいくつか述べておきましょう.

・spはプリインクリメントとする．すなわち〝*(++sp) = ～～″のように，まずスタックポインタをインクリメントした後にデータを与える.
・各関数はその呼び出された時点と終了時とでspを同一に保たねばならない．ただし，エラーによる大域脱出の時はspを無視して構わない.
・引数リスト，環境リストは，eval，applyがあらかじめスタックに積んでおくため，一般の関数では気にしなくてよい.

では，環境リストや引数リストの他にソフトスタックにはどんなものを積むべきでしょうか．Cのローカル変数でLispのオブジェクトを示すものはスタックに積んでおくべきなのですが，こ

れらをすべて積んでいたのではあっという間にスタックを使い尽くしてしまいます．スタックに積むのは必要最小限でなくてはなりません．すると次のような結論に達します．

・自分の内側でガベージコレクトの起こる可能性のない関数ではスタックは関係ない．やってこないガベージコレクトに備えるのは無意味である．したがって気を付けるのは，直接あるいは間接に自由領域からの取り出しを必要とする関数内である．自由領域からの取り出しにおいてのみガベージコレクトが起こる可能性がある．

・任意のリストはその頭だけがスタックにあればよい．したがって，環境や引数の一部を指しているようなCのローカル変数はスタックに積む必要はない．環境や引数はすでにスタックにあるからである．

・以上のことから，環境や引数の他にスタックに積む必要のあるのは，新しく製作中のリストである．これもその先頭だけ押さえていればよい(Fig. 6.2)．

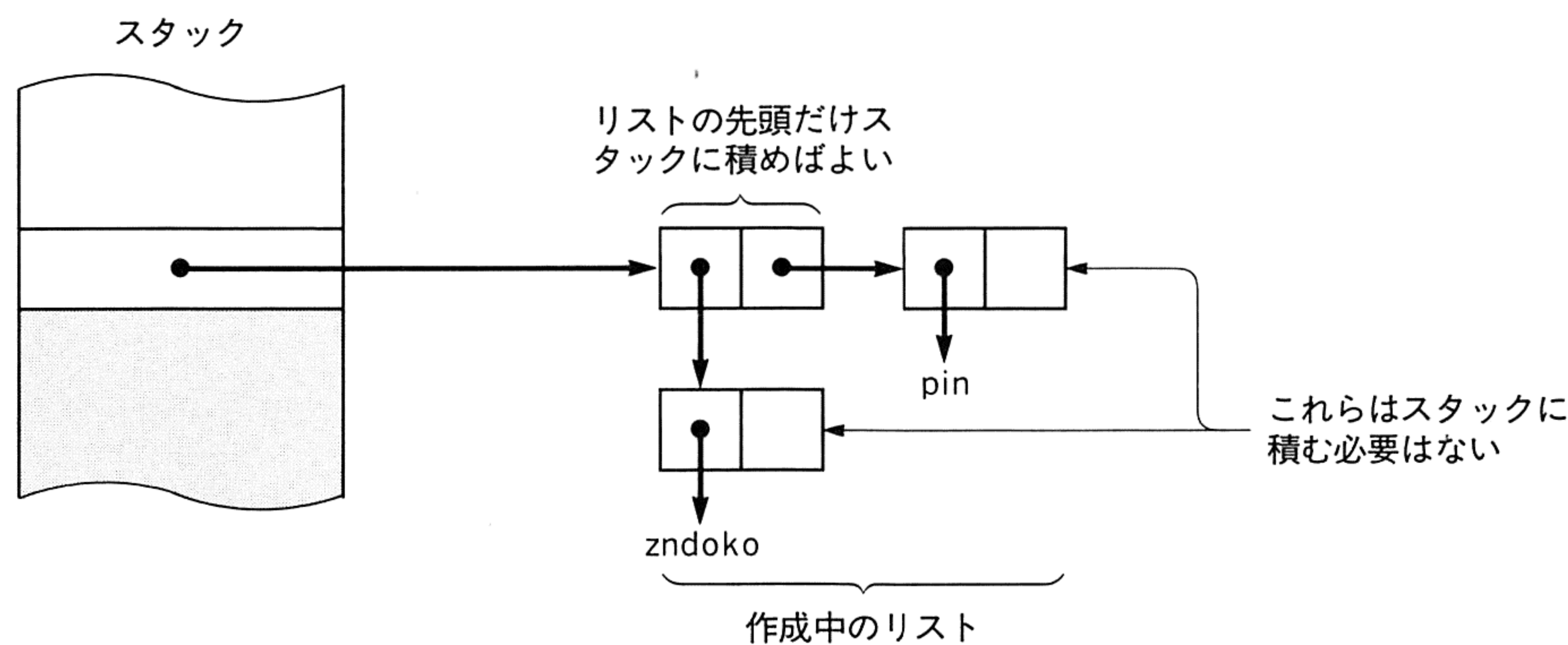



Fig. 6.2 頭を押さえる

では実際のプログラム上でソフトウェアスタックの使い方を見ることにしましょう．

6.1.4 ソフトウェアスタックの追加に伴うプログラムの変更

ソフトウェアスタックを使うためには，UNDEADバージョンのプログラムリストに，次のような変更が必要になります．

● lisp.h (List 6.1)

スタック領域の大きさに関する定数〝STACKSIZ〞と，ガベージコレクタが使用するftype用のマスク〝NONMRK〞が追加され，さらにソフトウェアスタックのオーバーフローをチェックするためのマクロ〝stackcheck〞が定義されます．また，ソフトウェアスタックのオーバーフローエラーのためのコード〝STACKUP〞が追加されます．

● defvar.h/var.h (List 6.2/List 6.3)

セルポインタへのポインタstacktop, spが追加されます. さらにガベージコレクタのメッセージを出すかどうかのフラグverbosが加わります.

● read.c (List 6.4)

リストを作る関数mk_list()にリストの先頭を退避するためのスタック処理が加わります. また, 文字列領域をアクセスするmk_sub()にガベージコレクタの呼び出しが付きます. ガベージコレクタgbc()については後述します.

● eval.c (List 6.6)

eval()には引数リストの退避がつき, apply()とbind()には環境リストの退避が, evallist()とpush()には新しく作る引数リストと環境リストの頭を押さえる動作が追加されます.

● error.c (List 6.7)

ソフトウェアスタックのオーバーフローが起きたというエラーメッセージがerr_msg配列の最後に追加されます.

● main.c (List 6.8)

init()の中にスタック領域の確保と初期化が加わり, main()内のループの中にspを初期設定する文が入ります.

```
sp = stacktop - 1;
```

となっているのは, 先に述べたとおりspがプリインクリメントだからです. 〝−1〞によりスタック領域を先頭から使うことができます.

6.1.5 ガベージコレクタの作成

いよいよガベージコレクタ本体の説明に入ることにしましょう. ガベージコレクタ本体の作成には, gbc.cファイルの大幅な変更が必要になります. UNDEADバージョンのgbc.cのファイルをList 6.5とすべて差し換えなくてはなりません.

● newcell(), newatom(), newnum() (List 6.5 13行〜48行)

いままでのUNDEADバージョンでは, 自由リストが空になったらすぐにエラーを返していましたが, その代わりにガベージコレクタの起動をします. ガベージコレクタがエラーを返したら,

未使用の構造体を集めるのに失敗したということです．gbc()の２つの引数の意味については後述します．

● gbc()　(List 6.5 50行～85行)

このガベージコレクタが起動すると，わずかの間Ｓ式の評価が止まります．この時間を少しでも短くするためにセル，シンボルアトム，数値，文字列の全領域を集めるのでなく，要請があった領域だけを集めることにします．ただし，セルだけはかならず集めます．またシンボルアトム領域と文字列領域については，１度起こったならその時は２つの領域を同時に集めてしまいます．この関係はTable 6.1のようになります．

		GCを呼んだ関数				
		newcell()	newnum()	newatom()	mk_sub()	reclaim_f()
集める領域	セル	○	○	○	○	○
	数値アトム	×	○	×	×	○
	シンボル	×	×	○	○	○
	文字列	×	×	○	○	○

Table 6.1　ガベージコレクタの集める領域

gbc()の第１引数nは数値領域を集めるかどうかのフラグ，第２引数aはシンボルアトム領域および文字列領域を集めるかどうかのフラグです．今の時点ではoblistに登録されないシンボルアトムというのがまだ存在しないため，不必要になるシンボルアトムはありません．ですからシンボルアトム領域のガベージコレクタは意味がありませんが，一応機能としてつけておきます．

まず，ガベージコレクタがメッセージを表示する状態(verbosの値がnil以外に設定されている状態)になっているなら，ガベージコレクタが起動したことを示すメッセージを表示します．その際，*reclaim*というLisp関数によりガベージコレクタが起動した時には，

"You suprised Gbc,"

と出力し，そうでなければ

"Gbc surprised You,"

と出力します．*reclaim*による起動かどうかは，Table 6.1のとおり，すべての領域を集めるのかどうかでわかりますが，これはフラグn，aがともに0でない時に相当します．

次に，oblistとソフトウェアスタックから到達できるすべてのセルに，関数mark()を用いてマークを付け，その後回収にかかります．セルはかならず集めて回り，数値とシンボルアトム，文

字は引数のフラグを見て集めるかどうかを決めます．これらのフラグは用がなくなれば，集めることのできた自由領域の大きさを表すのに使います．セルとシンボルアトムにはかならずマークされますが，これは集める作業の中で取り除かれます．したがって，シンボルアトム領域の塵集めをしなかった時は，シンボルアトムに付けられたマークを消す作業が必要になります．

最後に，集めた自由領域の量をメッセージとして出力します．

● mark()　(List 6.5 87行～112行)

構造体にマークを付ける関数です．マークをつける所は構造体のタイプを示すメンバidの最上位ビットです．普通はこのビットは0で空いており，使用中の構造体であることを示すためには，0x80とorをとって，ビットを立てればよいのです．

マークを付ける対象がnilにぶつかっていたら，そこで終わりで何もせずに戻ります．nilにはマークを付けません．すでにマーク済みのオブジェクトにぶつかった時も同じです．マークが付けられていなかったら，その構造体にマークを付け，そこからたどれるオブジェクトにマークを付けに行きます．

数値アトムはnフラグがONになっている時だけ印を付けますが，セルやシンボルアトムのようにそこから矢印が出ていくようなものは，かならず印を付けなくてはなりません．循環があると無限ループに陥るからです．シンボルアトムについては値(value)と属性リスト(plist)はかならずたぐっていきますが，関数ボディについては，Lispで定義された関数のみマークします．これはftypeをNONMRKでマスクをとればわかります．

● col_cell()，col_num()，col_atom()　(List 6.5 130行～214行)

この3つの関数については基本的な処理は同じです．つまり，集める領域の先頭から最後まで，マークが付いているものはマークを消し，マークのないものは自由リストへつないでいくのです．この時リストの最後はかならずnilにしておかなくてはなりません．この処理において，最初だけはつなぐリストがないので特別扱いになります．マークを消すのは0x80をビット反転したものとandをとればよいのです．では，これらの細かい差異について触れておきましょう．col_cell()では各セルのcarはかならずnilにしておきます．数値アトムでは，idを_FIXに直しておかねばなりません．シンボルアトムではnilにマークされていないことに注意します．nilはシンボルアトム領域の先頭にありますから，シンボルアトム領域の2番目から回収すればOKです．

また，これらの関数は集めた未使用構造体の数を数えておいてそれを返します．

● col_str()　(List 6.5 216行～240行)

文字列領域の回収をおこないますが，これはただ単に文字列を移動するのではだめで，それを指しているシンボルアトム内のポインタも書き換えねばなりません．したがって，シンボルアト

ム領域を調べながら進まねばなりません．

シンボルアトム内の印字名へのポインタが指しているのは，文字領域ではかならずヌル文字の次にある文字です．したがって，ヌル文字の次の文字に注目し，そこを指しているポインタがあるかどうか，使用中のシンボルアトムをすべて捜してみます．もし，そのようなポインタが見つかれば，その文字から次のヌル文字までは使用中の文字列であり，見つからなければ，その文字列はいらないことになります(Fig. 6.3)．

シンボルアトム領域
◎ cons　◎ car　マークがない abc　◎ lambda　マークがない Skara Brae
c o n s　c a r　a b c　l a m b d a　S k a r a ␣ B r a e
文字列領域
◎マークがないシンボルアトムが指している文字列は必要ない
◎マークがないシンボルアトムが指している文字列は必要ない

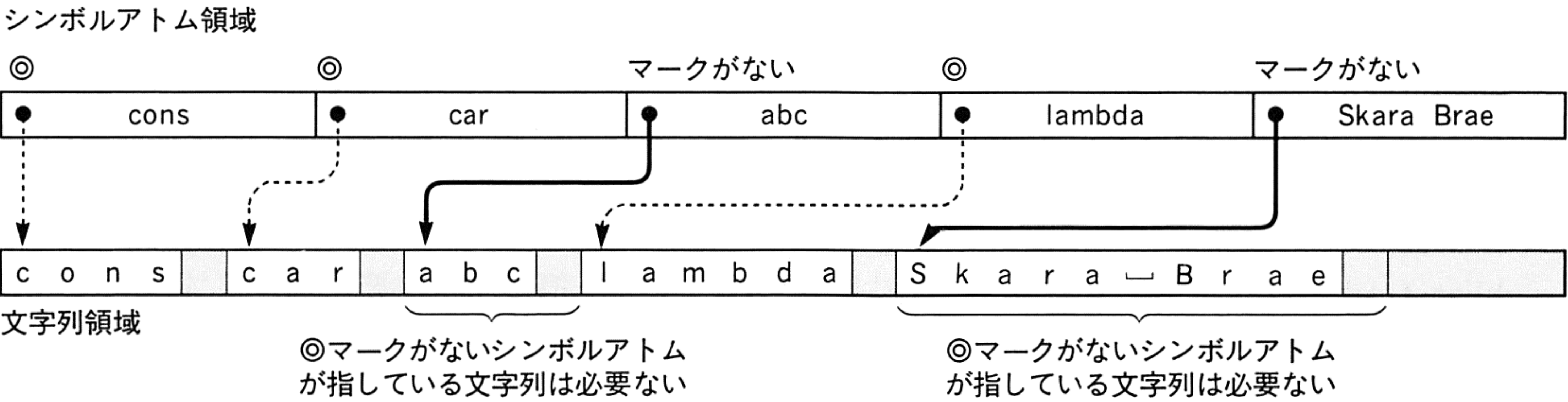



Fig. 6.3　シンボルアトムと文字列領域

プログラム上ではsとendという2つの文字ポインタによって検索と移動をおこないます．

sは現在未使用かどうかを検索中の文字ポインタで，endはガベージコレクトの終った領域の最後を指しています．sに等しい文字ポインタがシンボルアトム中に見つからなければ，sを次のヌル文字の次の文字まで進めて，再び検索します．sに等しいポインタが見つかった場合は，sからの文字列をendの指す所に移し，sとendを進めます．こうして，sがnewstr(文字領域の未使用部分の先頭)に達するまで繰り返します(Fig. 6.4)．なお，例によってnilは未使用ということはあり得ないので，検索の対象からはずしてあります．

最後に文字領域の空き部分の大きさを返します．

end　s　newstr
n i l　c o n s　a b c　d e f
ここまでOK　検索の結果未使用だった　これから調べる　ここまで調べる
abcが使われていた時
end　s　newstr
n i l　c o n s　a b c　a b c　d e f
copy

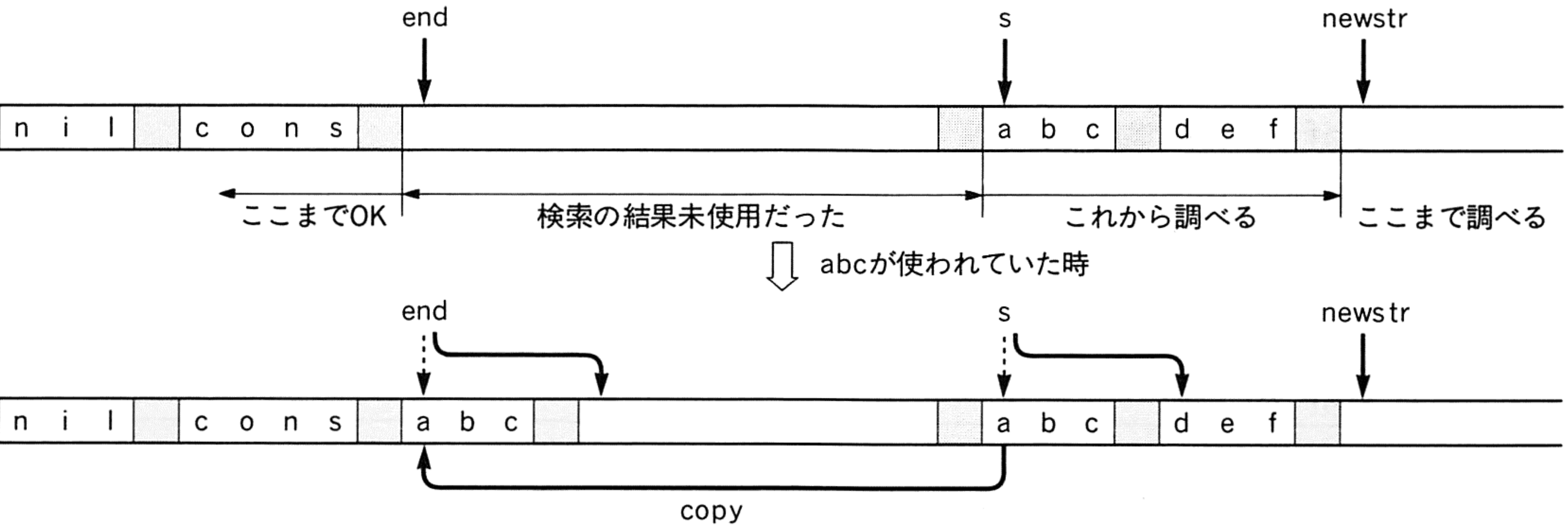



Fig. 6.4　文字列領域のガベージコレクト

6.1.6 ガベージコレクタに関連するLisp関数の追加・変更

ガベージコレクタに関連してreclaim_f(), verbos_f()の2つの関数がLisp上に増設されます. それに伴い, inisubr.cがList 6.9のように変更されます. また, ソフトウェアスタックの使用に関連して, putprop_f(), oblist_f()に変更が加わります.

● putprop_f() (List 6.10)

newcell()を呼び出す直前に, ソフトウェアスタックのチェックをおこないます.

● oblist_f() (List 6.11 1行〜25行)

oblist_f()がリストのコピーをするため, 製作中のリストを退避する必要があります. コピーは新しいセルを必要とするからです.

● reclaim_f() (List 6.11 27行〜31行)

ガベージコレクトをすべての領域に対して強制的に起こします.

● verbos_f() (List 6.11 33行〜47行)

引数がnilならメッセージの出力をOFFにし, そうでなければ, メッセージの出力をONにします. これは大域変数verbosによって切り替えられます.

List 6.1 lisp.h 追加

```
1: #define NONMRK   (_UD | _SR) ┐
2: #define USED     0x80        ├ マーク用マスク
3: #define FREE     (~USED)     ┘
4: #define STACKSIZ     0x0500       /* 1280 */
5: #define stackcheck  if(sp>=stacktop+STACKSIZ){error(STACKUP);return(NULL1);}
6: #define STACKUP 28  /* Softwear stack used up */
```

List 6.2 defvar.h 追加

```
1: int      err, err_no;
2: CELLP    *stacktop, *sp; ………ソフトスタック用変数
3: int      verbos;………………………ガベージコレクタのメッセージ用フラグ
```

List 6.3 var.h 追加

```
1: extern int       err, err_no;
2: extern CELLP     *stacktop, *sp;
3: extern int       verbos;
```

List 6.4 read.c mk_list(), mk_sub()を変更

```
 1: static CELLP mk_list(level)
 2: int level;
 3: {
 4:     char    mode;
 5:     CELLP   cp1, cp2;
 6:     CELLP   newcell(), error();
 7:
 8:     if (*txtp++ == '[') mode = SUP;
 9:     else                mode = NORM;
10:
11:     if (skipspace() == NULL)    return error(EOFERR);
12:     if (*txtp == ')') {
13:         ++txtp;
14:         return (CELLP)nil;
15:     }
16:     if (*txtp == ']') {
17:         if (mode == SUP || level == TOP)
18:             ++txtp;
19:         return (CELLP)nil;
20:     }
21:     if (*txtp == '.')               return error(PSEXP);
22:     stackcheck;
23:     cp1 = *++sp = newcell(); ec; ………リストの先頭をスタックに退避する
24:     getcar(cp1, UNDER); ec;
25:     if (skipspace() == NULL)    return error(EOFERR);
26:     while (*txtp != ')' && *txtp != ']') {
27:         if (*txtp == '.') {
28:             ++txtp;
29:             if (skipspace() == NULL)            return error(EOFERR);
30:             if (*txtp == ')' || *txtp == ']')   return error(PSEXP);
31:             getcdr(cp1, UNDER); ec;
32:             break;
33:         }
34:         cp2 = newcell(); ec;
35:         cp1->cdr = cp2;
36:         getcar(cp2, UNDER); ec;
37:         cp1 = cp2;
38:         if (skipspace() == NULL)    return error(EOFERR);
39:     }
40:     if (*txtp == ']')
41:         if (mode == NORM && level == UNDER) return *sp--;
42:     ++txtp;
43:     return *sp--;
44: }
45:
46: static ATOMP mk_sub(nam)
47: STR nam;
48: {
49:     int     length = strlen(nam) + 1;
50:     ATOMP   ap, newatom();
51:     STR     strcpy();
52:     CELLP   error();
53:
54:     if (newstr + length > strtop + STRSIZ) {
55:         gbc(OFF, ON);
56:         if (newstr + length > strtop + STRSIZ)
57:             return (ATOMP)error(STRUP);
58:     }
59:
60:     nam = strcpy(newstr, nam);
61:     newstr += length;
62:
63:     ap = newatom(); ec;
64:     ap->value = (CELLP)ap;
65:     ap->name = nam;
66:     ap->plist = (CELLP)nil;
67:     ap->ftype = _NFUNC;
68:     return ap;
69: }
```

List 6.5 gbc.c ファイルごと差し替える

```
 1: /*                                          */
 2: /*              GBC                         */
 3: /*                                          */
 4: /*  get new cell, atom, and num             */
 5: /*  gabage collect routine                  */
 6: /*                                          */
 7:
 8: #include    "lisp.h"
 9: #define     forever      for(;;)
10: #define     USED         0x80
11: #define     FREE         (~USED)
12:
13: CELLP newcell()
14: {
15:     CELLP   cp;
16:
17:     if (freecell == (CELLP)nil) {
18:         gbc(OFF, OFF); ec;
19:     }
20:     cp = freecell;
21:     freecell = freecell->cdr;
22:     cp->cdr = (CELLP)nil;
23:     return cp;
24: }
25:
26: ATOMP newatom()
27: {
28:     ATOMP   ap;
29:
30:     if (freeatom == nil) {
31:         gbc(OFF, ON); ec;
32:     }
33:     ap = freeatom;
34:     freeatom = (ATOMP)freeatom->plist;
35:     return ap;
36: }
37:
38: NUMP newnum()
39: {
40:     NUMP    np;
41:
42:     if (freenum == (NUMP)nil) {
43:         gbc(ON, OFF); ec;
44:     }
45:     np = freenum;
46:     freenum = freenum->value.ptr;
47:     return np;
48: }
49:
50: gbc(n, a) ………ガベージコレクタ   nは数値領域を集めるかどうかのフラグ
51: int n, a;                         aは文字アトム領域を集めるかどうかのフラグ
52: {
53:     CELLP   *spl;
54:     int     i, s;
55:
56:     if (verbos) {
57:         if (n & a)  fprintf(stdout, "¥nYou surprised Gbc,¥n");
58:         else        fprintf(stdout, "¥nGbc surprised You,¥n");
59:     }
60:
61:     for (i = 0; i < TABLESIZ; ++i)              ┐
62:         mark(oblist[i], n);                     │ oblistとソフトスタックから
63:     for (spl = stacktop; spl <= sp; ++spl)      │ 使用中の構造体をたぐっていく
64:         mark(*spl, n);                          ┘
65:
66:     i = col_cell(); ec;
67:     if (n) {
```

```
 68:         n = col_num();  ec;
 69:     }
 70:     if (a) {
 71:         s = col_str();
 72:         a = col_atom(); ec;
 73:     }
 74:     else
 75:         rem_mark_atom(); ………アトム領域のちり集めをしないときは文字アトム構造体に
 76:                              まだマークが残っているので消す
 77:     if (verbos) {
 78:         fprintf(stdout, "¥tfree cell = %d¥n", i);
 79:         if (n)  fprintf(stdout, "¥tfree num  = %d¥n", n);
 80:         if (a)  {
 81:             fprintf(stdout, "¥tfree atom = %d¥n", a);
 82:             fprintf(stdout, "¥tfree str  = %d¥n", s);
 83:         }
 84:     }
 85: }
 86:
 87: static mark(cp, n) ………使用中の構造体にマークをつける
 88: CELLP cp;              nは数値領域にマークをつけるかどうかのフラグ
 89: int n;
 90: {
 91:     char    c = cp->id;
 92:     CELLP   error();
 93:
 94:     if (cp == (CELLP)nil)   return; ………nilにはマークをつけない
 95:     if (c & USED)           return;
 96:     switch (c) {
 97:         case _ATOM: cp->id |= USED;
 98:                     mark(((ATOMP)cp)->value, n);
 99:                     mark(((ATOMP)cp)->plist, n);
100:                     if(!((((ATOMP)cp)->ftype) & NONMRK))  } Lispで定義された関数の
101:                         mark(((ATOMP)cp)->fptr, n);       } 場合にのみマークをつける
102:                     break;
103:         case _CELL: cp->id |= USED;
104:                     mark(cp->car, n);
105:                     mark(cp->cdr, n);
106:                     break;
107:         case _FIX:
108:         case _FLT:  if (n)  cp->id |= USED;
109:                     break;
110:         default:    return (int)error(ULO);
111:     }
112: }
113:
114: static rem_mark_num() ………数値領域につけられたマークを消す
115: {
116:     NUMP    np;
117:
118:     for (np = numtop; np < numtop + NUMSIZ; ++np)
119:         np->id &= FREE;
120: }
121:
122: static rem_mark_atom() ………文字アトム領域につけられたマークを消す
123: {
124:     ATOMP   ap;
125:
126:     for (ap = atomtop + 1; ap < atomtop + ATOMSIZ; ++ap)
127:         ap->id = _ATOM;
128: }
129:
130: static col_cell() ………セル領域のちり集め
131: {
132:     int n = 1;
133:     CELLP   end, cp = celltop;
134:
135:     while (cp->id & USED) { ………最初の未使用セルをさがす
```

```
136:         cp->id &= FREE;
137:         if (++cp >= celltop + CELLSIZ) {
138:             rem_mark_num();
139:             rem_mark_atom();
140:             return (int)error(CELLUP);
141:         }
142:     }
143:     freecell = end = cp++;
144:     end->car = (CELLP)nil;
145:
146:     for (; cp < celltop + CELLSIZ; ++cp) {
147:         if (cp->id & USED) {
148:             cp->id &= FREE;
149:             continue;
150:         }
151:         end->cdr = cp;
152:         end = cp;
153:         end->car = (CELLP)nil;
154:         ++n;
155:     }
156:     end->cdr = (CELLP)nil;
157:     return n;
158: }
159:
160: static col_num()
161: {
162:     int n = 1;
163:     NUMP    end, np = numtop;
164:
165:     forever {
166:         if (!(np->id & USED))   break;
167:         np->id &= FREE;
168:         if (++np >= numtop + NUMSIZ) {
169:             rem_mark_atom();
170:             return (int)error(NUMUP);
171:         }
172:     }
173:     freenum = end = np++;
174:     end->id = _FIX;
175:
176:     for (; np < numtop + NUMSIZ; ++np) {
177:         if (np->id & USED) {
178:             np->id &= FREE;
179:             continue;
180:         }
181:         end->value.ptr = np;
182:         end = np;
183:         end->id = _FIX;
184:         ++n;
185:     }
186:     end->value.ptr = (NUMP)nil;
187:     return n;
188: }
189:
190: static col_atom()
191: {
192:     int n = 1;
193:     ATOMP   end, ap = atomtop + 1;
194:
195:     forever {
196:         if (!(ap->id & USED))   break;
197:         ap->id &= FREE;
198:         if (++ap >= atomtop + ATOMSIZ)
199:             return (int)error(ATOMUP);
200:     }
201:     freeatom = end = ap++;
202:
203:     for (; ap < atomtop + ATOMSIZ; ++ap) {
```

（138〜140行）エラーで脱出する前に，GCがつけてしまったマークを消しておく（セル領域のマークは概に消してしまったのでこのままでよい）

（169〜171行）エラーで脱出する前に，GCがつけてしまったマークを消しておく（セル領域，数値領域のマークは，概に消してしまったので，このままでよい）

（193行）nilにはマークが付かないのでnilを除いてちり集めを行なう

```
204:         if (ap->id & USED) {
205:             ap->id &= FREE;
206:             continue;
207:         }
208:         end->plist = (CELLP)ap;
209:         end = ap;
210:         ++n;
211:     }
212:     end->plist = (CELLP)nil;
213:     return n;
214: }
215:
216: static col_str()
217: {
218:     STR     s, end;
219:     ATOMP   ap;
220:
221:     *newstr = '¥0';
222:     for (s = end = strtop + strlen("nil")+1; s < newstr;) { ………nilはちり集め
223:         for (ap = atomtop+1; ap < atomtop + ATOMSIZ; ++ap) {     から除く
224:             if (!(ap->id & USED))   continue;
225:             if (ap->name == s) {
226:                 if (end != s) {
227:                     strcpy(end, s);
228:                     ap->name = end;
229:                 }
230:                 while(*end++ != '¥0');
231:                 break;
232:             }
233:         }
234:         while (*s++ != '¥0');
235:         while (*s == '¥0' && s < newstr)
236:             ++s;
237:     }
238:     newstr = end;
239:     return (int)((strtop + STRSIZ) - newstr);
240: }
```

List 6.6 eval.c ファイルごと差し替える

```
  1: /*                                          */
  2: /*            EVAL and APPLY                */
  3: /*                                          */
  4:
  5: #include    "lisp.h"
  6:
  7: CELLP eval(form, env)
  8: CELLP form, env;
  9: {
 10:     CELLP   cp, apply(), atomvalue(), evallist(), error();
 11:     ATOMP   func;
 12:
 13:     switch (form->id) {
 14:         case _ATOM:
 15:             cp = atomvalue((ATOMP)form, env);
 16:             break;
 17:         case _FIX:
 18:         case _FLT:
 19:             return form;
 20:         case _CELL:
 21:             stackcheck;
 22:             ++sp;
 23:             func = (ATOMP)form->car;
 24:             if (eval_arg_p(func)) {
 25:                 *sp = evallist(form->cdr, env); ………引数リストはスタックに退避する
```

```
26:                     if (err) break;
27:                 }
28:                 else
29:                     *sp = form->cdr; ………引数リストはスタックに退避する
30:                 cp = apply((CELLP)func, *sp, env);
31:                 sp--;
32:                 break;
33:             default:
34:                 error(ULO);
35:     }
36:     if (err == ERR) {
37:         pri_err(form);
38:         return NULL;
39:     }
40:     return cp;
41: }
42:
43: static int eval_arg_p(func)
44: ATOMP func;
45: {
46:     if (func->id == _ATOM && func->ftype & _EA)
47:         return TRUE;
48:     if (func->id == _CELL && ((CELLP)func)->car == (CELLP)lambda)
49:         return TRUE;
50:     return FALSE;
51: }
52:
53: static CELLP apply(func, args, env)
54: CELLP func, args, env;
55: {
56:     CELLP   (*funcp)(), bodies, result = (CELLP)nil;
57:     CELLP   bind(), error();
58:     char    funtype;
59:
60:     switch (func->id) {
61:         case _ATOM:
62:             funtype = ((ATOMP)func)->ftype;
63:             if (funtype & _UD)
64:                 return error(UDF);
65:             if (funtype & _SR) {
66:                 funcp = (CELLP (*)())((ATOMP)func)->fptr;
67:                 if (funtype & _EA)
68:                     return (*funcp)(args);
69:                 else
70:                     return (*funcp)(args, env);
71:             }
72:             func = ((ATOMP)func)->fptr;
73:         case _CELL:
74:             if (func->cdr->id != _CELL)
75:                 return error(IFF);
76:             if (func->car == (CELLP)lambda) {
77:                 bodies = func->cdr->cdr;
78:                 stackcheck;
79:                 *++sp = bind(func->cdr->car, args, env); ec; ………環境の退避
80:                 for (; bodies->id == _CELL; bodies = bodies->cdr) {
81:                     result = eval(bodies->car, *sp); ec;
82:                 }
83:                 sp--;
84:                 return result;
85:             }
86:         default:
87:             return error(IFF);
88:     }
89: }
90:
91: static CELLP evallist(args, env)
92: CELLP args, env;
93: {
```

```
 94:     CELLP   cp1, newcell(), eval();
 95:
 96:     if (args->id != _CELL)
 97:         return (CELLP)nil;
 98:     stackcheck;
 99:     *++sp = newcell(); ec; ………作成中のリストの先頭はスタックに退避する
100:     cp1 = *sp;
101:     cp1->car = eval(args->car, env); ec;
102:     args = args->cdr;
103:     while (args->id == _CELL) {
104:         cp1->cdr = newcell(); ec;
105:         cp1 = cp1->cdr;
106:         cp1->car = eval(args->car, env); ec;
107:         args = args->cdr;
108:     }
109:     cp1->cdr = (CELLP)nil;
110:     return *sp--;
111: }
112:
113: CELLP bind(keys, values, env)
114: CELLP keys, values, env;
115: {
116:     CELLP   push(), error();
117:
118:     if (keys != (CELLP)nil && keys->id == _ATOM) {
119:         env = push(keys, values, env); ec;
120:         return env;
121:     }
122:     stackcheck;
123:     *++sp = env;
124:     while (keys->id == _CELL) {
125:         if (values->id != _CELL)
126:             return error(NEA);
127:         *sp = push(keys->car, values->car, *sp); ec;
128:         keys = keys->cdr;
129:         values = values->cdr;
130:     }
131:     if (keys != (CELLP)nil && keys->id == _ATOM) {
132:         *sp = push(keys, values, *sp); ec;
133:     }
134:     return *sp--;
135: }
136:
137: static CELLP push(key, value, env)
138: CELLP key, value, env;
139: {
140:     CELLP   newcell();
141:
142:     stackcheck;
143:     *++sp = newcell(); ec;
144:     (*sp)->cdr = env; ………作成中のリストの先頭はスタックに退避する
145:     env = *sp;
146:     env->car = newcell(); ec;
147:     env->car->car = key;
148:     env->car->cdr = value;
149:     return *sp--;
150: }
151:
152: static CELLP atomvalue(ap, env)
153: ATOMP ap;
154: CELLP env;
155: {
156:     CELLP   error();
157:
158:     while (env->id == _CELL) {
159:         if (env->car->id != _CELL)
160:             return error(EHA);
161:         if (env->car->car == (CELLP)ap)
```

```
162:            return env->car->cdr;
163:        env = env->cdr;
164:    }
165:    return ap->value;
166: }
```

List 6.7 error.c err_msg[]へメッセージを追加

```
  1:    "Software stack used up",
```

List 6.8 main.c toplevel_f(), reset_err(), init(), greeting()を変更

```
  1: toplevel_f()
  2: {
  3:     CELLP   *argp1, *argp2;
  4:     CELLP   read_s(), eval(), error();
  5:
  6:     argp1 = ++sp;
  7:     stackcheck;
  8:     argp2 = ++sp;
  9:
 10:     forever {
 11:         *argp1 = read_s(TOP);
 12:         if (*argp1 == (CELLP)eofread)   break;
 13:         ec;
 14:         *argp2 = eval(*argp1, (*argp2 = (CELLP)nil));
 15:         switch (err) {
 16:             case ERROK:
 17:             case ERR:   return (int)(err = ERROK);
 18:         }
 19:         print_s(*argp2, ESCON);
 20:         ec;
 21:         if (isatty(fileno(cur_fpi)))
 22:             fputc('¥n', cur_fpo);
 23:         fputc('¥n', cur_fpo);
 24:     }
 25:     sp -= 2;
 26: }
 27:
 28: static reset_err()
 29: {
 30:     cur_fpi = stdin;
 31:     cur_fpo = stdout;
 32:     err = NONERR;
 33:     txtp = oneline;
 34:     *txtp = '¥0';
 35:     for (sp = stacktop; sp < stacktop + STACKSIZ; ++sp) ………ソフトスタックの初期化
 36:         *sp = (CELLP)nil;
 37:     sp = stacktop - 1;
 38:     verbos = ON;
 39: }
 40:
 41: static init()
 42: {
 43:     char    *malloc();
 44:     int     quit();
 45:     int     i;
 46:     CELLP   cp;
 47:     ATOMP   ap;
 48:     NUMP    np;
 49:
 50:     freecell = celltop = (CELLP)malloc(sizeof(CELL) * CELLSIZ);
 51:     freeatom = atomtop = (ATOMP)malloc(sizeof(ATOM) * ATOMSIZ);
```

```
 52:     freenum = numtop = (NUMP)malloc(sizeof(NUM) * NUMSIZ);
 53:     newstr = strtop = (STR)malloc(STRSIZ);
 54:     sp = stacktop = (CELLP *)malloc(sizeof(CELLP) * STACKSIZ);
 55:
 56:     if (freecell == NULL || freeatom == NULL
 57:         || freenum == NULL || newstr == NULL || sp == NULL) {
 58:         printf("Oops! Alloc Error : Too Large Data Area.¥n");
 59:         printf("Please change --SIZ (defined in lisp.h).¥n");
 60:         exit(1);
 61:     }
 62:
 63:     nil = freeatom++;
 64:
 65:     for (cp = celltop; cp < celltop + CELLSIZ; ++cp) {
 66:         cp->id = _CELL;
 67:         cp->car = (CELLP)nil;
 68:         cp->cdr = cp + 1;
 69:     }
 70:     (--cp)->cdr = (CELLP)nil;
 71:
 72:     for (ap = atomtop + 1; ap < atomtop + ATOMSIZ; ++ap) {
 73:         ap->id = _ATOM;
 74:         ap->plist = (CELLP)(ap + 1);
 75:     }
 76:     (--ap)->plist = (CELLP)nil;
 77:
 78:     for (np = numtop; np < numtop + NUMSIZ; ++np) {
 79:         np->id = _FIX;
 80:         np->value.ptr = np + 1;
 81:     }
 82:     (--np)->value.ptr = (NUMP)nil;
 83:
 84:     for (i = 0; i < TABLESIZ; ++i)
 85:         oblist[i] = (CELLP)nil;
 86:
 87:     mk_sys_atoms();
 88:     ini_subr();
 89:     signal( SIGINT, quit );
 90: }
 91:
 92: static greeting()
 93: {
 94:     fprintf(stdout,"¥n");
 95:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
 96:     fprintf(stdout,"¥t           M A D I¥n");
 97:     fprintf(stdout,"¥t  Will o'Lisp Version 0.50¥n");
 98:     fprintf(stdout,"¥t        (C)  1986, Mar¥n¥n");
 99:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
100: }
```

List 6.9 inisubr.c ファイルごと差し替える

```
 1: /*                                           */
 2: /*                  INISUBR                  */
 3: /*                                           */
 4: /*  initialize SUBR and FSUBR type function  */
 5: /*                                           */
 6:
 7: #include "lisp.h"
 8:
 9: ini_subr()
10: {
11:     init0();
12:     init1();
13: }
14:
```

```
15: static init0()
16: {
17:     CELLP car_f(), cdr_f(), cons_f();
18:     CELLP atom_f(), eq_f(), equal_f();
19:     CELLP quote_f(), de_f(), cond_f();
20:     CELLP setq_f(), oblist_f(), quit_f();
21:     CELLP putprop_f(), get_f(), remprop_f();
22:     CELLP read_f(), terpri_f();
23:     CELLP print_f(), prinl_f(), princ_f();
24:     CELLP minus_f(), plus_f();
25:
26:     defsubr("car",       car_f,       _SUBR);
27:     defsubr("cdr",       cdr_f,       _SUBR);
28:     defsubr("cons",      cons_f,      _SUBR);
29:     defsubr("atom",      atom_f,      _SUBR);
30:     defsubr("eq",        eq_f,        _SUBR);
31:     defsubr("equal",     equal_f,     _SUBR);
32:     defsubr("quote",     quote_f,     _FSUBR);
33:     defsubr("de",        de_f,        _FSUBR);
34:     defsubr("cond",      cond_f,      _FSUBR);
35:     defsubr("setq",      setq_f,      _FSUBR);
36:     defsubr("oblist",    oblist_f,    _SUBR);
37:     defsubr("quit",      quit_f,      _SUBR);
38:     defsubr("putprop",   putprop_f,   _SUBR);
39:     defsubr("get",       get_f,       _SUBR);
40:     defsubr("remprop",   remprop_f,   _SUBR);
41:     defsubr("read",      read_f,      _SUBR);
42:     defsubr("terpri",    terpri_f,    _SUBR);
43:     defsubr("print",     print_f,     _SUBR);
44:     defsubr("prinl",     prinl_f,     _SUBR);
45:     defsubr("princ",     princ_f,     _SUBR);
46:     defsubr("minus",     minus_f,     _SUBR);
47:     defsubr("plus",      plus_f,      _SUBR);
48: }
49:
50: static init1()
51: {
52:     CELLP   reclaim_f(), verbos_f();
53:
54:     defsubr("reclaim",  reclaim_f,  _SUBR);
55:     defsubr("verbos",   verbos_f,   _SUBR);
56: }
57:
58: static defsubr(name, funcp, type)
59: STR name;
60: CELLP (*funcp)();
61: char type;
62: {
63:     ATOMP   ap, mk_atom();
64:
65:     ap = mk_atom(name); ec;
66:     ap->ftype = type;
67:     ap->fptr = (CELLP)funcp;
68: }
```

List 6.10　fun.c putprop_f()を変更

```
 1: CELLP putprop_f(args)
 2: CELLP args;
 3: {
 4:     CELLP   val, cp, error();
 5:     ATOMP   key, ap;
 6:
 7:     if (args->id != _CELL
 8:         || args->cdr->id != _CELL
 9:         || args->cdr->cdr->id != _CELL)
10:         return error(NEA);
11:     if ((ap = (ATOMP)args->car)->id != _ATOM
```

```
12:         || (key = (ATOMP)args->cdr->cdr->car)->id != _ATOM)
13:         return error(IAA);
14:     val = args->cdr->car;
15:     cp = ap->plist;
16:     for (cp = ap->plist; cp->id == _CELL; cp = cp->cdr->cdr) {
17:         if ((ATOMP)cp->car == key)
18:             return (cp->cdr->car = val);
19:     }
20:     stackcheck;
21:     *++sp = newcell(); ec;
22:     cp = *sp;
23:     cp->car = (CELLP)key;
24:     cp->cdr = newcell(); ec;
25:     cp->cdr->car = val;
26:     cp->cdr->cdr = ap->plist;
27:     ap->plist = *sp--;
28:     return val;
29: }
```

List 6.11 control.c oblist_f()を変更，reclaim_f()，verbos_f()を追加

```
 1: CELLP   oblist_f()
 2: {
 3:     int     i = 0;
 4:     CELLP   cp1, cp2, cp3, newcell();
 5:
 6:     stackcheck;
 7:     cp1 = *++sp = newcell(); ec;
 8:     for (i = 0; i<TABLESIZ; ++i)
 9:         if ((cp2 = oblist[i]) != (CELLP)nil) break;
10:     for(; cp2->cdr != (CELLP)nil; cp2 = cp2->cdr) {
11:         cp1->car = cp2->car;
12:         cp3 = newcell(); ec;
13:         cp1->cdr = cp3;
14:         cp1 = cp3;
15:     }
16:     cp1->car = cp2->car;
17:     for (++i; i<TABLESIZ; ++i)
18:         for(cp2 = oblist[i]; cp2 != (CELLP)nil; cp2 = cp2->cdr) {
19:             cp3 = newcell(); ec;
20:             cp1->cdr = cp3;
21:             cp1 = cp3;
22:             cp1->car = cp2->car;
23:         }
24:     return *sp--;
25: }
26:
27: CELLP reclaim_f()
28: {
29:     gbc(ON, ON); ec;
30:     return (CELLP)nil;
31: }
32:
33: CELLP verbos_f(arg)
34: CELLP arg;
35: {
36:     if (arg == (CELLP)nil)
37:         if (verbos) return (CELLP)t;
38:         else        return (CELLP)nil;
39:     if (arg->car == (CELLP)nil) {
40:         verbos = OFF;
41:         return (CELLP)nil;
42:     }
43:     else {
44:         verbos = ON;
45:         return (CELLP)t;
46:     }
47: }
```

6.2 機能拡張(2) 制御構造の作成 —MALORバージョン—

ここでの拡張により，プログラミングの自由度が飛躍的に増大します．Lispは原則的には関数型言語ですが，早くから手続き型言語風の仕様を採り入れてきました．さらに最近のLispは大域脱出やエラートラップの機構を持つようになっています．これらの機能を使うことにより，プログラムの制御を任意の場所に任意の時点で思いのままに移すことができます．現在Lispで書かれるプログラムのほとんどはこれらの機能に全面的に依存しています．

6.2.1 prog 形式

prog は，手続き型言語風の制御構造を提供する，強力で，たいへん便利な fsubr 関数ですが，反面，*prog* は Lisp の暗黒面の一端をも担っています．ともあれ，まず *prog* の機能を説明しましょう．

prog は，基本的には与えられた引数を前から順に評価する関数で，nil を値として返します．

```
% (prog () (prin1 pin) (setq pin 'tan) (prin1 pin))
pintannil

% pin
tan
```

第1引数は特別な意味を持ちますが，ここでは nil になっています．第2引数の評価により pin の値(オートクォートによりその値も pin)が出力されます．その次の引数が評価されると pin の値が tan になり，その次の引数が評価されると，pin の値 tan が出力されて，*prog* の実行が終わり，*prog* の値 nil が返ってきています．なお，*prog* の外で pin の値を見てみるとちゃんと tan になっています．

これだけの機能ならば，Will o'Lisp には lambda 式が複数の本体を実行する機能(implicit progn の機能)がありますから，*prog* の価値はあまりないのですが，*prog* にはこの他に次の3つの大きな役割があります．

(1) ローカル変数の設定
(2) *go* 式による評価順序の流れの変更
(3) *return* 式による値返し

以下，Cのプログラムと比較しながらこれらの機能をまとめて説明していくことにします．

Lispではある処理を繰り返すのには，原則的には再帰呼び出しを使います．たとえば，1からnまでの和(1+2+ …… +n)を求めるには，

```
(de sumupto (n)
      (cond ((zerop n) 0)
            (t (plus n (sumupto (difference n 1))))))
```

という関数*sumupto*を定義します．この関数はnが0の時は0を返し，それ以外の時は*sumupto*を再帰呼び出ししてn−1の時の値(すなわち1からn−1までの和)を計算させ，それにnを足したものを返すことで求める和を得ています．

しかし，手続き型言語に慣れた人には，次のような考え方のCの関数の方がピンとくるのではないでしょうか．

```
int sumupto(n)
int n;
 {
    int sum = 0;
    while (n > 0) {
        sum = sum + n;
        n--;
    }
    return(sum);
}
```

*prog*は，Lispでもこういった手続き的な考え方でプログラムを書きたいという要求を満たすために用意されました．*prog*を使って*sumupto*を書いてみると次のようになります．

```
(de sumupto (n)
      (prog ((sum 0))
            loop (cond ((equal n 0) (return sum)))
                 (setq sum (plus sum n))
                 (setq n (difference n 1))
                 (go loop)))
```

説明しなくてもだいたいの見当はつくでしょう．*prog*の第1引数

((sum 0))

ではローカル変数sumを宣言し，その初期値として0を与えています．カッコが2重になっているのを不思議に思う人がいるかもしれませんが，これはローカル変数をひとつしか宣言していないためで，一般には*prog*の第1引数では複数のローカル変数を宣言でき，

((<ローカル変数> <初期値>) (<ローカル変数> <初期値>) …… (<ローカル変数> <初期値>))

のようになります．このリストをローカル変数リストと呼ぶことにします．

その次の引数loopはラベルで，一番最後の引数(go loop) の中で，飛び先を指定するのに使われています．*go*はCやBASICのgoto文のようなもので，*go*式は*prog*の中でのみ使用できるようになっています．ラベルには任意のアトムを使用できます．逆にいうと，*prog*の引数がアトムならば，それはラベルとみなされるわけです．ラベルは評価されません．

第1引数とアトムを除いた引数は，前から順に評価されます．そしてその中に*go*式と*return*式を置くことができます．*go*式は先に述べたようにgoto文のようなもので，

(go <ラベル>)

の形をしており，これが評価されるとラベルで指定された場所の次の引数から評価が続けられます．*sumupto*の例では，(go loop)によって再びcondから順に評価がおこなわれることになります．

一方return式は，

(return <S式>)

の形で使われ，*prog*の評価をそこで終了し，そのS式を評価した値を*prog*全体の評価値として返す働きを持ちます．*sumupto*の中では，condの中にあり，sumを返すのに使われています．

したがって，*prog*版の*sumupto*は，ラベルの次のcondでループ終了条件としてnが0になったかどうかを見て，0ならばsumを返し，まだ0でなければsumにsum+nを代入し，nの値を1へらす，という以上の手続きを繰り返すプログラムとなっているのです．

6.2.2 ローカル変数の有効範囲

ローカル変数リストには，先の例のように変数と初期値のリスト，または，変数のみを置くことができます．変数のみ書いた時はその初期値はnilになります．たとえば，

(prog ((pin nil) (tan nil)) ……)

と書く代わりに次のように書くことができます．

(prog (pin tan) ……)

ここで宣言されたローカル変数は，直接 *prog* の中ではなく，*prog* の中で使われている(f)expr 関数の中からでも参照できます．

```
(de pin () (prin1 sug))
```

と関数 pin を定義しておくと，次のようになります．

```
% (prog ((sug 'tan)) (pin))
tannil
```

prog のローカル変数として sug が宣言されて初期値 tan が与えられている状態で関数 pin を呼ぶと pin の中の sug は，*prog* のローカル変数として参照されるため，tan が出力されます．nil は，*prog* が返した値です．

また，ひとつ下のレベルだけでなく，*prog* に呼ばれた関数に呼ばれた関数の中のように，ローカル変数はネストの深いところからでも参照できます．ただし，ネストの途中で，別の *prog* があって，そこで同じアトムがローカル変数として再宣言されている場合は，そのアトムについては内側の *prog* の中からは外側の *prog* のローカル変数の値はアクセスできません．

```
% (de zdo () (prin1 jun)
%            (prog ((jun 2))
%                  (prin1 jun)
%                  (setq jun 3)
%                  (prin1 x)))
zdo
```

と関数 zdo を定義しておくと，次のような結果が得られます．

```
% (prog ((jun 1)) (prin1 jun) (zdo) (prin1 jun))
11231nil
```

まず，トップレベルで入力している *prog* のローカル変数リストで jun がローカル変数として宣言され初期値 1 が与えられ，その直後の prin1 によりそれが出力されています．次に関数 zdo が呼ばれると，その中の prin1 で出力される jun は，外側の *prog* のローカル変数であり，やはり 1 が出力されます．次に内側の *prog* に入ってそのローカル変数リストで jun がローカル変数として再宣言され，初期値 2 が代入されます．内側の *prog* の中では外側の *prog* の値は見えなくなり，したがって内側の *prog* の最初の prin1 は 2 を出力します．次の setq で jun に 3 が代入されますがこの jun も内側の *prog* のローカル変数です．その次の prin1 によってそれが出力された後，内側の *prog* を抜けると，内側の *prog* のローカル変数の値は存在しなくなり，外側の *prog* のそれが再び

見えるようになります．ゆえに，この後 zdo を抜けて，次の prin1 によって出力されるのは，外側の *prog* のローカル変数 jun の値 1 になります．

6.2.3 *go* ラベルの有効範囲と *return* の脱出先

go ラベルの有効範囲もローカル変数の場合とほぼ同様に考えることができます．ラベルは *prog* の直接の引数でなければなりませんが，そのラベルにジャンプする *go* 式は，*prog* の中でありさえすれば，いくらネストの深いところにあってもかまいません．たとえば，

```
% (de tam (n)
%       (cond ((zerop n) (go loktofeit))
%             (t (tam (difference n 1)))))
tam
```

という関数を定義します．これは，n が 0 になるまで再帰呼び出しを繰り返し，0 になったらいきなり loktofeit というラベルにジャンプする関数です．この関数はその外にラベル loktofeit を含んだ *prog* の存在を仮定しており，*prog* がなければ，エラーを起こします．

```
% (tam 3)

Opps !
Error No. 29 : No Such go-label.
At toplevel
```

しかし，必要なラベルを含んだ *prog* がありさえすれば，どんなにネストが深いところからでも *go* 式によって脱出が可能です．

```
% (prog () (tam 10) loktofeit (princ |I returned to castle.|))
I returned to castle.nil
```

一方，*prog* の中に別の *prog* があって，内側の *prog* に同名のラベルが存在した場合ジャンプ先は内側のラベルになり，外側まで脱出することはできません．

```
% (de castle ()
%       (prog ()
%             (prog ()
%                   (tam 3)
%                 loktofeit
%                   (princ |I'm at one level under castle.|)
```

```
%                  (go malor))
%           loktofeit
%             (princ |It's incredible but I'm at castle! |)
%           malor))
castle

%(castle)
I'm at one level under castle.nil
```

関数 *castle* を評価させると，すぐに2つの *prog* の内側に入り込み，関数 tam が呼び出されます．tam は3回再帰呼び出しをした後ラベル loktofeit へジャンプします．この時，2つの *prog* のそれぞれが loktofeit というラベルを持ちますが，内側の *prog* のそれにしか到達することはできません．したがって〝I'm at one level under ... 〞というメッセージを出力した後，次の(go malor)によって再びジャンプし，外側の *prog* のラベル malor に達して *castle* の処理を終了します．

return 式も，同様な状況にあります．いくつかの *prog* がネストしている場合，*return* 式の脱出先は，その *return* 式より外側で，最も近い *prog* となり，それより外側の *prog* までは脱出できません．

```
% (de nue (n)
%       (cond ((zerop n) (return))
%             (t (nue (difference n 1)))))
nue
```

関数 *nue* は n が 0 ならば *return* し，それ以外の時は，n が 0 になるまで再帰呼び出しを繰り返します．*nue* は *prog* の内側で使われることを前提としており，*prog* の外で評価するとエラーを起こします．

```
% (nue 3)

Opps!
Error No. 29 : No Such go-label.
At toplevel

% (de castle2 ()
%       (prog ()
%             (prog () (nue 10))
%             (princ |I'm at one level under castle. |))
%       (princ |I'm at castle! |) nil)
castle2
```

```
%(castle2)
I'm at one level under castle.I'm at castle ! nil
```

*castle2*を起動すると，まず2つの*prog*の内側に入り，(nue 10)が評価されます．(nue 10)は再帰呼び出しを繰り返した後*return*式で内側の*prog*を脱出します．したがって内側の*prog*の次にあるprincが評価されてから外側の*prog*が終了し，次にその後ろのprincが評価されます．そのため，2つのメッセージは，両方とも出力されることになります．

ここまでの説明でわかるように，*prog*の*go*式や*return*式を使うと，見事なスパゲティ・プログラムを作ることができ，他人にはまったく理解できない関数を容易に作成できます．*prog*がLispの暗黒面を担っているとはこういう意味です．ここは，あの有名な二人の会話を強く心に思い起こしておきましょう．

「暗黒面(だあくさいど)のほうが強いの？」
「いや，暗黒面のほうが容易なのじゃよ.」

6.2.4 let ―progの分解

*prog*は変数束縛と，*go*によるプログラムの流れの制御，*return*によるプログラムブロックの値返し，という3つの機能を含んでいます．このため*prog*はたいへん強力な関数になっていますが，場合によっては強力すぎて重いと感じることもあります．そこで最近のLispでは，各機能を独立させてそれぞれに関数がひとつ用意されています．Common Lisp仕様書では，変数束縛は*let*，*go*は*tagbody*，*return*は*block*という関数でサポートされるようになっています．Will o'Lispではこのうち*let*を採り入れてみました．*let*は，*prog*のようにローカル変数リストを持ちますが，その中で*go*や*return*を使うことはできません．引数はimplicit prognとして実行され，したがって*let*の評価値はいちばん最後の引数の評価値になります．

```
% (let ((stock zndoko) zakkai)
%      (print stock)
%      (setq zakkai 'orange))
zndoko
orange

% zakkai
zakkai
```

上の例では stock と zakkai がローカル変数として lambda-bind され，stock には初期値 zdoko が与えられています．*let* の中で *setq* がおこなわれた場合，その値は環境リストの方に置かれます．*let* の評価が終わる時に，この lambda-bind は解除されます．したがってグローバル変数としての zakkai の値は影響されていません．

let の仲間で，*let* ＊という関数もあります．これは，ほとんど *let* と同じ機能を有しますが，*let* ではローカル変数リストで宣言された変数が並列に bind されるのに対し，*let* ＊では逐次 bind される点が異なっています．ローカル変数リストの中の順番は *let* では bind される値に何の影響も与えませんが，*let* ＊では，前から順に bind されるため，先におこなわれた bind によって後からおこなわれる bind の値が異なる可能性があるわけです．

```
% (let ((a 'b) (c a)) c)      ……①
a

% (let * ((a 'b) (c a)) c)    ……②
b
```

①ではローカル変数 a, b の bind は並行しておこなわれる（実際はもちろん順番にですが，初期値の評価環境に let の評価が始まる前のものを使う）ため，a の値は b に，c の値は a の値の（オートクォートによって）a になっています．②では，b が a の初期値として与えられた後，その環境の下で c の初期値 a が評価されるので，c には b が bind されています．

6.2.5 大域脱出

大域脱出とは，関数呼び出しのネストの深いところから通常の関数の処理の終了手続きを経由せずにネストの浅いところに制御を移すことをいいます．エディタなど，いくつかのモードを持つプログラムでの中断処理やモード遷移などにはどうしても欲しい機能です．

Lisp ではこれを *catch* および *throw* という 2 つの fsubr 関数で実現します．*catch* と *throw* はそれぞれ次のように引数をとります．

```
(catch  <タグ>  <本体1>  ……  <本体n>)      ……①
(throw  <タグ>  <本体1>  ……  <本体n>)      ……②
```

catch は本体を評価し，何事もなければその評価値を返します．もし本体の中で②のような throw が評価されると，throw の本体が順に評価され，次に大域脱出が起こります．この脱出は，throw のタグと同じタグを持つ catch まで行われ，本体 n の評価値が脱出先の catch の評価値として返されます．

```
% (catch 'second (print 'strike) (print 'ball) (print 'ball))      ……①
strike
ball
ball
ball

% (de steal () (throw 'second 'Out ! ))                            ……②
steal

% (catch 'second (print 'strike) (steal) (print 'ball))            ……③
strike
Out !
```

①では *catch* の引数が順番に評価されているだけで，*catch* は *progn* の働きしかしていません．*catch* の値は，いちばん後ろの引数である(print 'ball)の評価値 ball になっています．ところが，③では，②のように中で *throw* を呼んでいる関数を *catch* の中に置くと，*throw* のところで *catch* の実行は終了し，*catch* の評価値も *throw* が投げてきた〝Out !〞になっています．

大域脱出は *prog* の *go* や *return* を使ってもおこなえます．*catch* と *throw* の場合はタグを使って脱出先を指定すると同時に値を返しているので，*go* と *return* の働きを同時におこなっているようなものです．しかし，*catch* と *throw* を使う場合には，大域脱出であることが明示され，プログラムの読みやすさにいくぶん貢献することにもなります．

6.2.6 エラートラップ

エラーによってプログラムの実行を中断されたくないことがあります．たとえば，アクセスしようとするファイル名を入力させて，ユーザーが存在しないファイル名を入力した時，そのままアクセスして，エラーを起こしたまま終わってしまう場合，あるいはまた大きなプログラムで0による除算が起こった時，ただちに終了せず，そこに到達するまでに得た途中結果を表示するような場合を考えてください．特に実用的プログラムにおいては，どんなエラーにも耐え得る頑丈な機構が必要となります．Will o'Lisp では，この要求を満たすため，*catcherror* という関数を備えました．*catcherror* は，引数を implicit progn として実行し，何事もなければ nil を返します．引数の中でエラーが発生した場合は，実行は中断し，*catcherror* の値は発生したエラーのエラーナンバーになります．

```
% (prog nil (catcherror (car 'b)) (print ok ! ))

Oops !
```

```
Error No.13 : Illegal arguement--List required
At (car (quote b))

ok !
nil
```

このようにエラーメッセージの後，実行は継続されています．

6.2.7 prog 機能に関連する関数の追加・変更

これらの制御構造を追加するにあたって，MADI バージョンのプログラムリストに prog.c ファイルが追加され，さらに次のような変更が必要となります．

● lisp.h (List 6.12)

エラーフラグ err を利用した大域脱出機構のため，err に与える 3 種類の定数が追加になります．また，5 種類のエラーコードが追加されます．

● defvar.h/var.h (List 6.13/List 6.14)

catch&throw のラベルおよび値の受け渡しに用いる大域変数が追加されます．

● error.c (List 6.15)

追加された 5 種類のエラーに対するエラーメッセージが err_msg [] の最後に追加されます．

● main.c (List 6.16)

throw/go/return により返されてきた値を受け取るための機構が toplevel_f()の中に追加されます．

● inisubr.c (List 6.17)

新しく追加される Lisp 関数の定義のため，関数 init2()が追加され，ini_subr()が変更になります．

● progn_f() (List 6.18 7 行～18 行)

これは implicit progn の語源で，その名のとおりの働きをします．

● prog_f(), go_f(), ret_f() (List 6.18 20 行～104 行)

prog はひとつで 3 種もの機能を持つ関数なので，そのプログラムはかなり長いものになってい

ますが，それぞれの機能を実現している部分に分けて見れば，いたって簡単なものです．

この関数では，ソフトウェアスタックを2つ使っています．最初でspを2つふやし，そのspについて，＊spを環境リストに，＊(sp−1)をgoのラベルリストに使います．

まず，ローカル変数の実現です．「5.5 S式の評価」の時に，applyの中で関数の仮引数と実引数をlambda-bindするbind()という関数を使いました．ローカル変数を実現するには，評価と並行してbind()の働きをおこないます．すなわち，ローカル変数リストから順に変数とその値を取り出し，値を評価し，これを変数とlambda-bindしていきます．

その次のwhileは，goラベルの登録をおこなっています．ここでは*prog*の本体を頭から調べていき，アトムがあったらラベルとみなして，それ以降の本体を含むリストすべてをgoラベルリストにconsしています．これをもしS式で出力すると非常に大きなリストになりますが，メモリ内ではポインタで同じところを指しているだけなので，何の問題もないのです．たとえば，

```
(prog ()
      first (go second)
      second (go third)
      third (print x))
```

というprogのgoラベルリストは，

```
((first (go second) second (go third) third (print x))
 (second (go third) third (print x))
 (third (print x)))
```

のようになりますが，このリストはFig. 6.5のような構造をしているわけです．

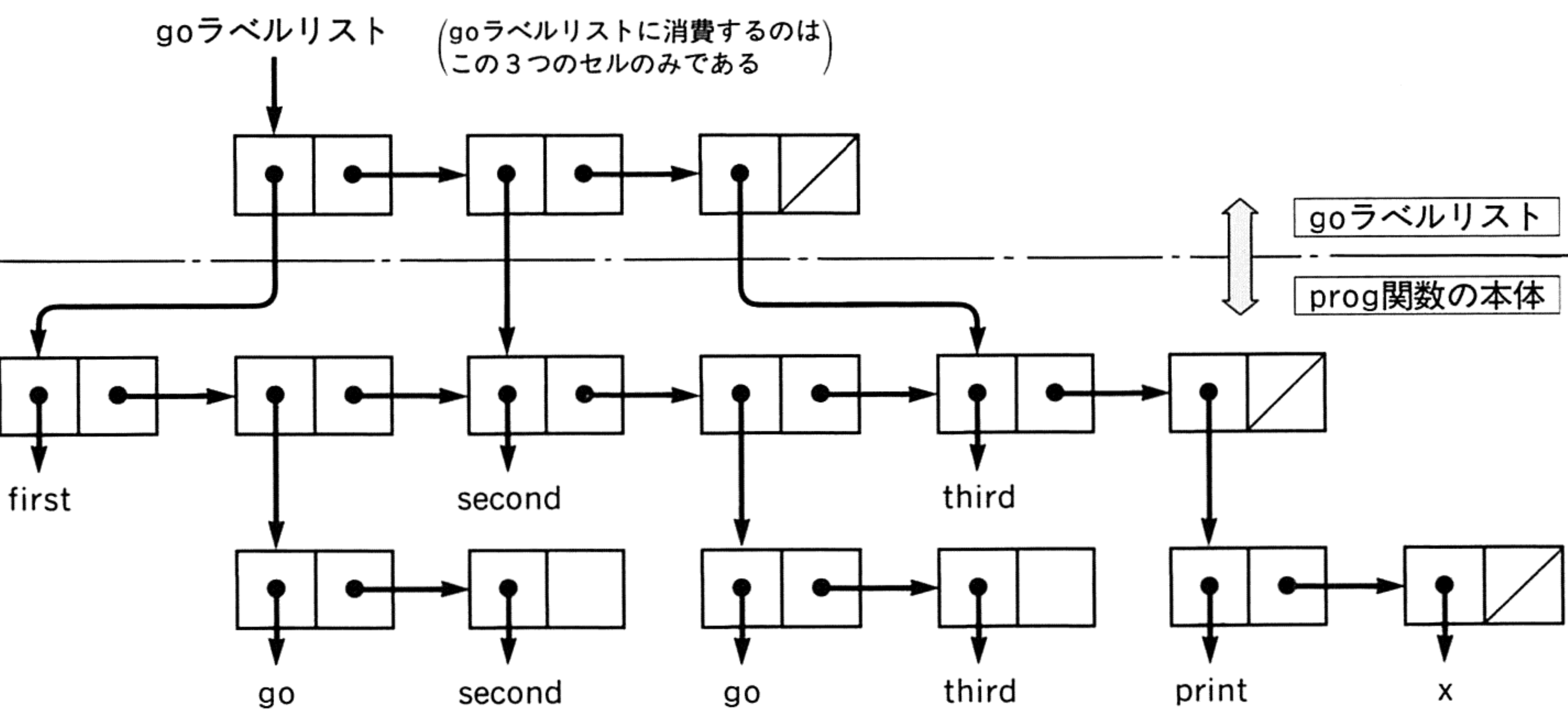



Fig. 6.5 goラベルリスト

この後に続くのが本体の逐次評価のループです．しかしながらその前に，その時点での sp の値を保存しておきます．これは *go* や *return* によって sp の値がずれた時に，修正するための準備です．さて，ただの implicit progn と異なっているのは，以下の点です．

① 本体がアトムの時は評価しないでただスキップする．

② 本体を評価した時に変数 err の値が GO ならば，*go* の処理をおこなう．

③ 同様に err の値が RET ならば，*return* の処理をおこなう．

err という変数は通常エラーが起きているかどうかを表し，通常は 0 になっています．エラーが発生した時には，この変数が 1 にセットされます．するとプログラムのあちこちにちりばめられたマクロ ec によってトップレベルまでの脱出がおこなわれるのでした(「5.6 エラー処理」参照)．ところで，脱出がおこなわれるためには，err の値が 0 でさえなければよかったのです．ここではこれを利用して，err を 1 以外の値に設定して脱出を発生させ，これによって実行を中断し，*go* と *return* の処理をおこなうようにしたわけです．GO と RET は lisp.h で定義された定数です．

さて，go_f()の方では，err の値を GO に設定し，ラベルをグローバル変数 throwlabel においています．このラベルは脱出によって prog_f()に戻った後，*go* ラベルリストから一致するラベルを探し出すのに使います．ここで使っている assoc()は連想リストという一種の辞書的なデータを検索する関数で，go ラベルを連想リストとみなした場合，ラベルに対応するリストは，そのラベルに続いて逐次評価するべき本体リストになっています．したがって，prog_f()に戻ってから，まず，sp と err をもとに戻し，ラベルに対応するリストを取り出して評価中のリストに置き換えてやればよいわけです．

また同様に ret_f()でも，err の値を RET に設定し，今度は返すべき値を throwlabel においています． prog_f()に戻ってからは sp と err を修正し， prog_f()の値として throwlabel を返しています．

● catch_f()，cerr_f()，throw_f()　(List 6.18 106行〜178行)

これらは，prog_f()の *go* や *return* と同じようにグローバル変数 err と throwlabel を使って実現しています．catch_f()と throw_f()ではさらにグローバル変数として throwval を使い，これで値の受け渡しをおこなっています．cerr_f()は *catcherror* の定義です．これも catch_f()とほぼ同様ですが，〝Pseudo S expression(S 式でないものを読み込んだ)〟エラーの場合には，テキストポインタの初期化もおこなっています．

● let_f()，lets_f()　(List 6.18 180行〜247行)

この 2 つは，prog_f()のローカル変数束縛の部分だけを取り出したものです．

List 6.12 lisp.h 追加

```
1: #define THROW    2 ………throwの受け先を探している状態を表す
2: #define GO       3 ………goの行き先を探している状態を表す
3: #define RET      4 ………returnの戻り先を探している状態を表す
4:
5: #define NSG      29  /* No such go-label */
6: #define RWP      30  /* Return without Prog */
7: #define TTA      31  /* Throw Tag must be an Atom */
8: #define TWC      32  /* Throw without Catch */
9: #define ILV      33  /* Illegal Local Variable List */
```

List 6.13 defvar.h 追加

```
1: CELLP   throwlabel, throwval;
```

List 6.14 var.h 追加

```
1: extern CELLP    throwlabel, throwval;
```

List 6.15 error.c err_msg[]へメッセージを追加

```
1:      "No such go-label",
2:      "Return without Prog",
3:      "Throw Tag must be an Atom",
4:      "Throw without Catch",
5:      "Illegal Local Variable List",
```

List 6.16 main.c toplevel_f(), reset_err(), greeting()を変更

```
 1: toplevel_f()
 2: {
 3:     CELLP   *argp1, *argp2;
 4:     CELLP   read_s(), eval(), error();
 5:
 6:     argp1 = ++sp;
 7:     stackcheck;
 8:     argp2 = ++sp;
 9:
10:     forever {
11:         *argp1 = read_s(TOP);
12:         if (*argp1 == (CELLP)eofread)   break;
13:         ec;
14:         *argp2 = eval(*argp1, (*argp2 = (CELLP)nil));
15:         switch (err) {
16:             case ERROK:
17:             case ERR:    return((err = ERROK));
18:             case THROW: return (int)error(TWC);
19:             case GO:     return (int)error(NSG);
20:             case RET:    return (int)error(RWP);
21:         }
22:         print_s(*argp2, ESCON);
23:         ec;
24:         if (isatty(fileno(cur_fpi)))
25:             fputc('¥n', cur_fpo);
26:         fputc('¥n', cur_fpo);
27:     }
```

```
28:     sp -= 2;
29: }
30:
31: static reset_err()
32: {
33:     cur_fpi = stdin;
34:     cur_fpo = stdout;
35:     err = NONERR;
36:     txtp = oneline;
37:     *txtp = '¥0';
38:     for (sp = stacktop; sp < stacktop + STACKSIZ; ++sp)
39:         *sp = (CELLP)nil;
40:     sp = stacktop - 1;
41:     verbos = ON;
42:     throwlabel = throwval = (CELLP)nil;
43: }
44:
45: static greeting()
46: {
47:     fprintf(stdout,"¥n");
48:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
49:     fprintf(stdout,"¥t           M A L O R¥n");
50:     fprintf(stdout,"¥t  Will o'Lisp Version 0.60¥n");
51:     fprintf(stdout,"¥t       (C)  1986, Mar¥n¥n");
52:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
53: }
```

List 6.17 inisubr.c ini_subr()を変更，init2()を追加

```
 1: ini_subr()
 2: {
 3:     init0();
 4:     init1();
 5:     init2();
 6: }
 7:
 8: static init2()
 9: {
10:     CELLP progn_f(), throw_f(), catch_f();
11:     CELLP cerr_f(), let_f(), lets_f();
12:     CELLP prog_f(), ret_f(), go_f();
13:
14:     defsubr("progn",    progn_f,    _FSUBR);
15:     defsubr("throw",    throw_f,    _FSUBR);
16:     defsubr("catch",    catch_f,    _FSUBR);
17:     defsubr("catcherror",   cerr_f, _FSUBR);
18:     defsubr("let",      let_f,      _FSUBR);
19:     defsubr("let*",     lets_f,     _FSUBR);
20:     defsubr("prog",     prog_f,     _FSUBR);
21:     defsubr("return",   ret_f,      _SUBR);
22:     defsubr("go",       go_f,       _FSUBR);
23: }
```

List 6.18 prog.c 新しいファイルを作成

```
 1: /*                              */
 2: /*          Prog                */
 3: /*                              */
 4:
 5: #include    "lisp.h"
 6:
 7: CELLP progn_f(args, env)
 8: CELLP args, env;
```

```
 9: {
10:     CELLP   result, eval();
11:
12:     result = (CELLP)nil;
13:     while (args->id == _CELL) {
14:         result = eval(args->car, env); ec;
15:         args = args->cdr;
16:     }
17:     return result;
18: }
19:
20: CELLP prog_f(args, env)
21: CELLP args, env;
22: {
23:     CELLP varlist, forms, result, *currentsp, cp;
24:     CELLP bind(), eval(), error(), cons(), assoc();
25:
26:     if (args->id != _CELL)
27:         return error(NEA);
28:     if ((varlist = args->car)->id != _CELL && varlist != (CELLP)nil)
29:         return error(ILV);
30:     sp += 2;
31:     stackcheck;
32:     *sp = env;
33:     while(varlist->id == _CELL) { ……………………ローカル変数のbind
34:         if (varlist->car->id == _ATOM) { ………初期値なしの場合
35:             *sp = bind(varlist->car, (CELLP)nil, *sp); ec;
36:         }
37:         else if (varlist->car->id == _CELL) { ………初期値が与えられている場合
38:             *sp = bind(varlist->car->car, (CELLP)nil, *sp); ec;
39:             if (varlist->car->cdr->id != _CELL)
40:                 return error(ILV);
41:             (*sp)->car->cdr = eval(varlist->car->cdr->car, env);
42:             ec;
43:         }
44:         else
45:             return error(IAAL);
46:         varlist = varlist->cdr;
47:     }
48:     *(sp-1) = (CELLP)nil;
49:     forms = args->cdr;
50:     while (forms->id == _CELL) { ………goラベルをgoラベルリストに登録
51:         if (forms->car->id == _ATOM) {
52:             *(sp-1) = cons(forms, *(sp-1)); ec;
53:         }
54:         forms = forms->cdr;
55:     }
56:     forms = args->cdr;
57:     currentsp = sp;
58:     result = (CELLP)nil;
59:     while (forms->id == _CELL) {
60:         if (forms->car->id == _ATOM) {
61:             forms = forms->cdr;
62:             continue;
63:         }
64:         eval(forms->car, *sp);
65:         if (err == GO) {
66:             sp = currentsp;
67:             if (!(cp = assoc(throwlabel, *(sp-1))) || cp->id != _CELL)
68:                 return NULL;
69:             err = 0;
70:             forms = cp;
71:         }
72:         else if (err == RET) {
73:             sp = currentsp-2;
74:             err = 0;
75:             return throwval;
76:         }
```

```
 77:         ec;
 78:         forms = forms->cdr;
 79:     }
 80:     sp -= 2;
 81:     return (CELLP)nil;
 82: }
 83:
 84: CELLP go_f(args, env)
 85: CELLP args, env;
 86: {
 87:     CELLP error();
 88:
 89:     if (args->id != _CELL)
 90:         return error(NEA);
 91:     err = GO;
 92:     return(throwlabel = args->car);
 93: }
 94:
 95: CELLP ret_f(args)
 96: CELLP args;
 97: {
 98:     CELLP error();
 99:     err = RET;
100:
101:     if (args->id != _CELL)
102:         return(throwval = (CELLP)nil);
103:     return(throwval = args->car);
104: }
105:
106: CELLP catch_f(args, env)
107: CELLP args, env;
108: {
109:     CELLP   bodies, result, *cur_sp, error(), eval();
110:
111:     if (args->id != _CELL)
112:         return error(NEA);
113:     stackcheck;
114:     *++sp = eval(args->car, env); ec;
115:     if ((*sp)->id != _ATOM)
116:         return error(TTA);
117:     bodies = args->cdr;
118:     result = (CELLP)nil;
119:     throwlabel = throwval = (CELLP)nil;
120:     cur_sp = sp;
121:     while (bodies->id == _CELL) {
122:         result = eval(bodies->car, env);
123:         if (err == THROW && throwlabel == *cur_sp) {
124:             sp = --cur_sp;
125:             err = NONERR;
126:             return throwval;
127:         }
128:         ec;
129:         bodies = bodies->cdr;
130:     }
131:     sp--;
132:     return result;
133: }
134:
135: CELLP cerr_f(args, env)
136: CELLP args, env;
137: {
138:     CELLP   result, *cur_sp, eval();
139:     NUMP    np, newnum();
140:
141:     cur_sp = sp;
142:     while (args->id == _CELL) {
143:         result = eval(args->car, env);
```

```
144:         if (err == ERR || err == ERROK) {
145:             sp = cur_sp;
146:             err = NONERR;
147:             if (err_no == PSEXP)    *txtp = '¥0';
148:             np = newnum();  ec;
149:             np->value.fix = (long)err_no;
150:             return (CELLP)np;
151:         }
152:         args = args->cdr;
153:     }
154:     return  (CELLP)nil;
155: }
156: 
157: CELLP throw_f(args, env)
158: CELLP args, env;
159: {
160:     CELLP   bodies, result, error(), eval();
161: 
162:     if (args->id != _CELL)
163:         return error(NEA);
164:     stackcheck;
165:     *++sp = eval(args->car, env); ec;
166:     if ((*sp)->id != _ATOM)
167:         return error(TTA);
168:     throwlabel = *sp--;
169:     bodies = args->cdr;
170:     result = (CELLP)nil;
171:     while (bodies->id == _CELL) {
172:         result = eval(bodies->car, env); ec;
173:         bodies = bodies->cdr;
174:     }
175:     throwval = result;
176:     err = THROW;
177:     return NULL;
178: }
179: 
180: CELLP let_f(args, env)
181: CELLP args, env;
182: {
183:     CELLP   list, bodies, error(), bind(), eval();
184: 
185:     if (args->id != _CELL)
186:         return error(NEA);
187:     if ((list = args->car)->id != _CELL && list != (CELLP)nil)
188:         return error(ILV);
189:     stackcheck;
190:     *++sp = env;
191:     while (list->id == _CELL) {
192:         if (list->car->id == _ATOM) {
193:             *sp = bind(list->car, (CELLP)nil, *sp); ec;
194:         }
195:         else if (list->car->id == _CELL) {
196:             if (list->car->cdr->id != _CELL)
197:                 return error(ILV);
198:             *sp = bind(list->car->car, (CELLP)nil, *sp); ec;
199:             (*sp)->car->cdr = eval(list->car->cdr->car, env); ec;
200:         }
201:         else
202:             return error(IAAL);
203:         list = list->cdr;
204:     }
205:     bodies = args->cdr;
206:     while (bodies->id == _CELL) {
207:         list = eval(bodies->car, *sp); ec;
208:         bodies = bodies->cdr;
209:     }
210:     sp--;
211:     return list;
```

```
212: }
213:
214: CELLP lets_f(args, env)
215: CELLP args, env;
216: {
217:     CELLP   list, bodies, error(), bind(), eval();
218:
219:     if (args->id != _CELL)
220:         return error(NEA);
221:     if ((list = args->car)->id != _CELL && list != (CELLP)nil)
222:         return error(ILV);
223:     sp += 2;
224:     stackcheck;
225:     *sp = env;
226:     while (list->id == _CELL) {
227:         if (list->car->id == _ATOM) {
228:             *sp = bind(list->car, (CELLP)nil, *sp); ec;
229:         }
230:         else if (list->car->id == _CELL) {
231:             if (list->car->cdr->id != _CELL)
232:                 return error(ILV);
233:             *(sp-1) = eval(list->car->cdr->car, *sp); ec;
234:             *sp = bind(list->car->car, *(sp-1), *sp); ec;
235:         }
236:         else
237:             return error(IAAL);
238:         list = list->cdr;
239:     }
240:     bodies = args->cdr;
241:     while (bodies->id == _CELL) {
242:         list = eval(bodies->car, *sp); ec;
243:         bodies = bodies->cdr;
244:     }
245:     sp -= 2;
246:     return list;
247: }
248:
249: CELLP assoc(key, alist)
250: CELLP key, alist;
251: {
252:     while (alist->id == _CELL) { ………goラベルリストを検索する
253:         if (alist->car->id != _CELL)
254:             return error(IASSL);
255:         if (alist->car->car == key)
256:             return alist->car;
257:         alist = alist->cdr;
258:     }
259:     return NULL;
260: }
```

6.3 機能拡張(3) File I/Oの追加
―CALFOバージョン―

先の拡張でガベージコレクタとprog関数が搭載されたとはいっても，このLispはまだまだ力不足です．まずリスト処理関数が足りません．入出力関数がありません．数値演算も不十分です．しかしながら，これらのうちのいくつかは，すでにある関数を使って定義することができます．とはいうものの，毎回関数定義を入力するのはたいへんですから，少なくともプログラムのロード機能は欲しいところです．そこで，ガベージコレクタの作成とprog関数の追加に続く機能の整備として，入出力の拡張をすることにします．

6.3.1 Will o'Lisp の File I/O

Will o'Lispでのファイル記述の方法は以下のとおりです．

- ファイルを開く時は関数 *open* にファイル名を印字名に持つシンボルアトムを与える．
- *open* はオープンしたファイルに対し番号をつけ，その数値を返す．
- 入出力関数は *open* で得られた数値を引数にして入出力の対象を指定する．

内部では，複数のファイルをアクセスするため，C言語上でファイルポインタの配列をあらかじめ用意し，Lispの上ではその配列のインデックスでファイルを表します．このインデックスが *open* の返す値になります．以後このファイル番号ををファイルディスクリプタと呼びます．ファイルディスクリプタは配列の添字であるという性質上，0以上の整数になります(Fig. 6.6)．

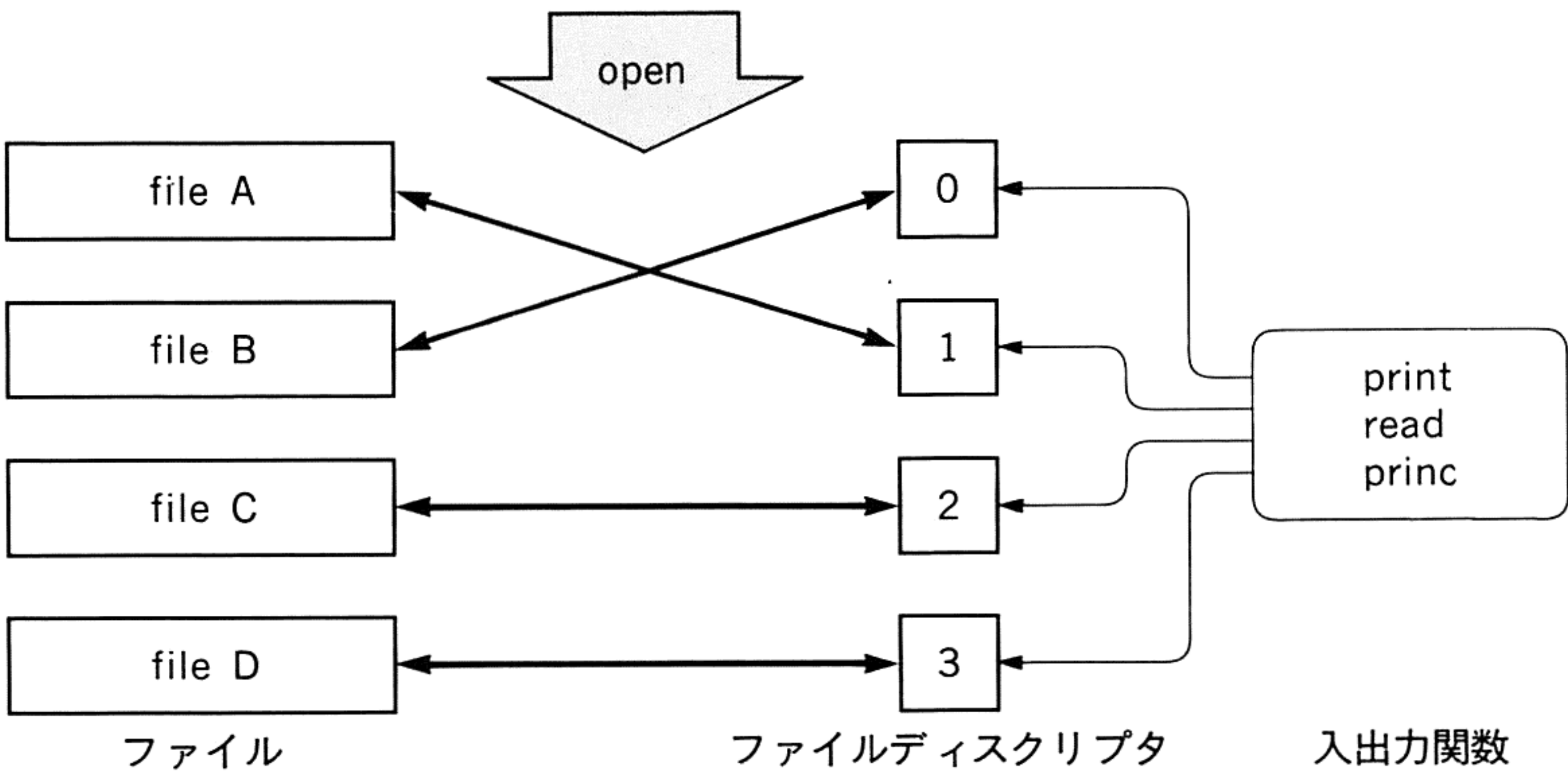



Fig. 6.6 入出力の概念

ファイルアクセスに関して例をあげましょう．"Welcome！¥n"という内容を持ったguild.lspというファイルを読むためには，

```
% (open'|guild.lsp|'r)   ……guild.lspを読むために開く
3                        ……ファイルディスクリプタ3が返る

% (read 3)               ……guild.lspから読み込む
Welcome！                ……guild.lspの内容
```

のように入力関数*read*に対して，どこから読み込むかをファイルディスクリプタで指定します．

このようにCALFOバージョンでは，これまでコンソールに対してのみしかおこなえなかったreadやprintを，ファイルディスクリプタで指定したファイルに対しておこなえるように拡張します．

●オープン時のモード (List 6.26 11行〜14行)

ファイルをオープンする時には，そのファイルに対してどういうアクセスをするのかを指定する必要があります．この指定は関数*open*にモードを示す文字列を与えることによっておこなわれますが，これに対応して，C上でfopen()に与えるために，モードを指示する文字列はmodestr[]という配列に登録してあります．この配列はファイルiofunc.cの先頭で定義してあります．この配列内の文字列とそのインデックスとの関係は以下のようになります(Table 6.2)．

インデックス(バイナリ表示)	文字列	モード
0 (000)	"a"	アペンド(テキスト)
1 (001)	"w"	ライト(テキスト)
2 (010)	"r"	リード(テキスト)
3 (011)	――	――
4 (100)	"ab"	アペンド(バイナリ)
5 (101)	"wb"	ライト(バイナリ)
6 (110)	"rb"	リード(バイナリ)

Table 6.2 FILE I/Oのモード

3番目にあたる文字列がありませんが，これはインデックスの第2ビットにバイナリのデータを扱うかどうかを表すフラグの意味を持たせるためです．

●ファイルポインタの配列 (List 6.19 1行〜4行)

Will o'Lispはファイル入出力にfopen()などのCの高レベルの入出力関数を使います．これらの関数群はファイルポインタを介して入出力をおこないますので，複数のファイルにアクセス す

るために，あらかじめファイルポインタ用の配列を用意してあります．

この配列からファイルのアクセスのモードを知ることができる必要があります．このモードは構造体 FILE の中にありますが，この構造体の仕様は各コンパイラによって異なる可能性があるので，先の modestr のインデックスにあたる数字と構造体 FILE を組にして新しい構造体を作り，それを配列としています(Fig. 6.7).

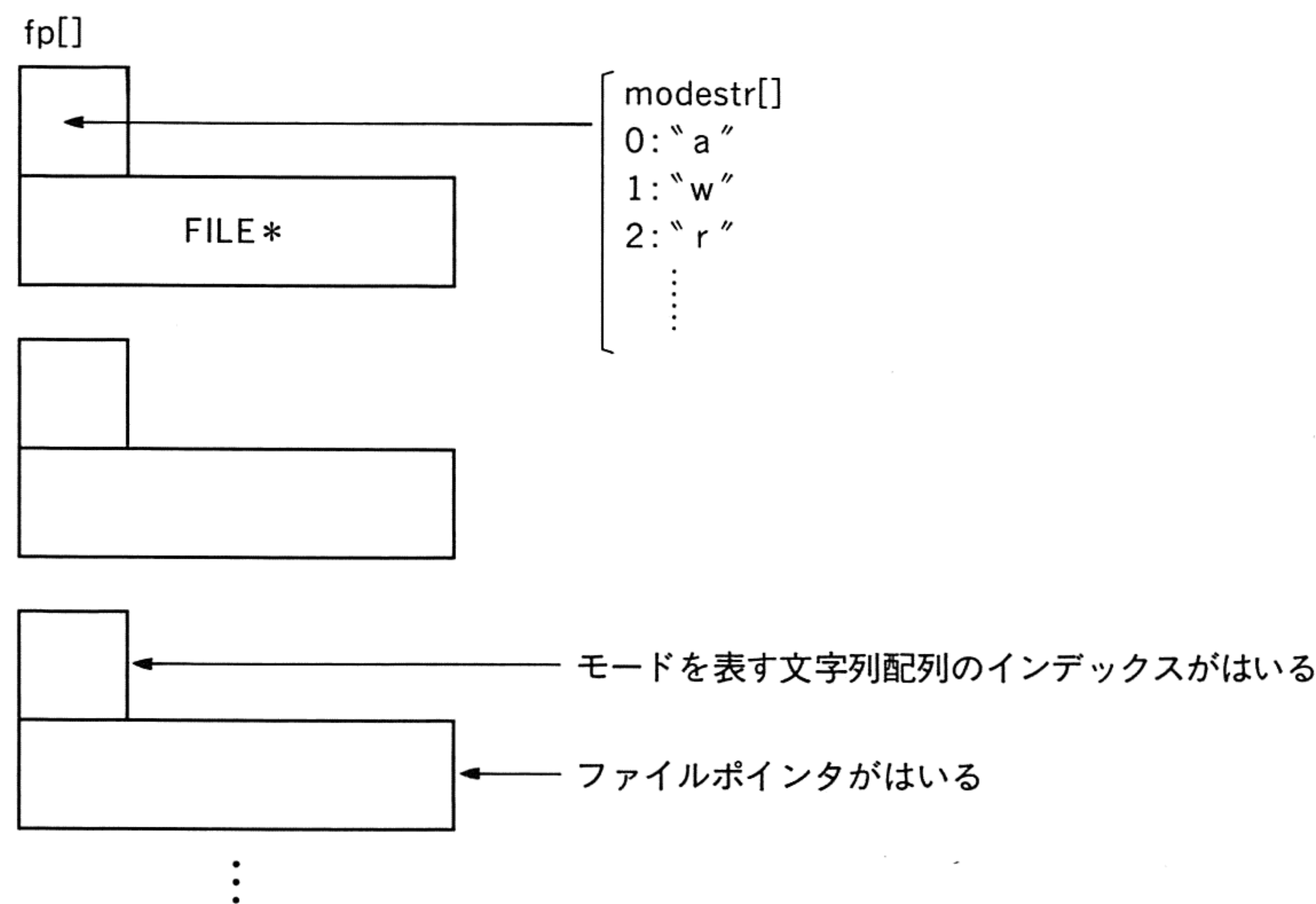



Fig. 6.7　ファイルポインタ構造体配列

i 番目のファイルのモードを知るには fp[i].mode とすればよく，また，modestr[fp[i].mode] とすれば，そのモードにあたる文字列が得られます．この配列の要素で，まだファイルと結びついていないものはファイルポインタの部分を NULL にし，未使用であることを表します．この初期化はファイル main.c の中の関数 init()の中にあります．

●標準入出力

先の配列要素の内，初めの3つは常に標準入出力に向けておきます．

0	:	標準入力	stdin
1	:	標準出力	stdout
2	:	標準エラー出力	stderr

したがって，ユーザーが使うことのできるのはファイルディスクリプタは3番以降になります．このファイルポインタの配列の大きさは lisp.h の中で "NFILES" として定義してありますが，これは少なくとも3以上でなくてはなりません．

6.3.2 オープン，クローズ

● open_f() (16行〜51行)

関数 *open* の引数は1個または2個です．第1引数はオープンするファイルの名前を印字名に持つシンボルアトムで，第2引数はオープンのモードを表すシンボルアトムですが，これは省略可能です．省略した場合はデフォルトのテキストリードのモードになります．第1引数のチェックの後，第1引数のシンボルアトムの印字名を関数 getfname()を使い(後述)ファイルネーム用の文字配列にコピーします．第2引数が省略された時はオープンのモードを示す文字列を "r" にし，さもなくばその文字列を第2引数のシンボルアトムの印字名にセットします．引数のチェックが終わると，ファイルポインタの配列上の空きを捜します．この時，0から2の要素はすでに標準の入出力ファイルにセットされているので，3以降の要素の検索をおこないます．空きの要素が見つかったらモードを示す文字列をチェックしますが，ここではその文字列からそのモードを数値化します．先ほど modestr [] について述べたとおり，バイナリモードでオープンする時にはモードを表す数値の第2ビットをONにすればよいのです．これでお膳立てができたので，実際に fopen()を試み，結果をファイルポインタ配列上にセットし，そのセットした場所のインデックスを関数 *open* の返す値として，数値アトムを作ります．

● getfname() (53行〜68行)

シンボルアトムの印字名からファイル名として文字列を指定された文字配列に複写します．単に strcpy()を使うだけにしないのは，拡張子が省略された場合，デフォールトとして ".LSP" を補うからです．初めに，シンボルアトムの印字名の長さに関係なく，複写先の文字列の長さ引くことの5文字分をコピーしてしまいます．余計な所までコピーしてしまいますが，シンボルアトムの長さより，複写先の大きさの方が優先されねばならないからです．最後に拡張子のピリオドを捜し，無ければデフォールトの ".LSP" を付け足しておきます．拡張子のないファイルをオープンする場合は最後にピリオドだけ付けておきます．

● close_f() (70行〜91行)

まず，引数をチェックし与えられたファイルディスクリプタを取り出します．そのファイルがオープンされていれば，クローズの処理をおこないますが，現在の入(出)力先となっているファイルを閉じる場合もありますので，そのような時は，その入(出)力先を標準入(出)力に戻しておきます．さらに，その処理が入力先の場合は，入力用の文字列バッファをクリアするために，文字バッファ内の入力文字を示すポインタの位置に文字列の終わりを示すヌル文字を入れて，リーダにバッファが空になったことを教えておきます．最後に，実際に fclose()の処理をおこない，閉じたファイルディスクリプタを値として返します．

6.3.3 基本出力関数

● print_f(), prin1_f(), princ_f() (List 6.26 93行〜152行)

これらの関数について FILE I/O を追加するためには，共通のサブルーチン prin()を変更するだけです．ファイルディスクリプタが省略された時は以前と同じですが，その指定がある時は，そのファイルディスクリプタが確保した配列の中に収まる適正値であることと，ファイルが読み込み用でないことを確認した後，出力先を一時そのファイルに変更して出力をおこないます．読み込み用かどうかは，Fig. 6.4 のとおり，タイプを表すコードの第0ビットと第1ビットがともに ON であることからわかります．出力先の変更はファイルポインタ cur_fpo に希望するファイルのファイルポインタを代入すればよいのです．

● terpri_f() (List 6.26 41行〜152行)

プログラム上は改行出力をするだけで，prin()と大差ありませんが，関数の返す値が異なります．*prin?*(？は1，c，tのいずれか)では，出力したオブジェクトを関数の値としましたが，この関数 *terpri* は nil を返します．

6.3.4 基本入力関数

● read_f() (List 6.26 154行〜170行)

関数 read は任意の入力ファイルからＳ式の入力をおこないます．指定されたファイルに入力を向けて，リーダを呼べばよいのですが，注意することがひとつだけあります．Will o'Lisp では，Ｓ式の入力を文字列単位で取り込んでおこないます．この文字列取り込み用のバッファは各ファイルについて用意されているのではなく，共通のバッファがひとつあるだけです(プログラム中では oneline [] となっている)．このことはある場合に支障をきたします．実際，Will o'Lisp は1回の入力で複数のＳ式を入力することが許されていますが，これが曲者です．たとえば，

```
% (open (quote |oracle|) (quote r))          ……ファイル oracle.lsp のオープン
3

% (setq message (read 3)) (car message)      ……2つのＳ式の入力
(The conjunction of the moons finds link !)  ……oracle.lsp の内容で，(setq)が返した値

The          ……(car message) の評価の結果

%            ……次の入力待ち
```

このような処理が期待されますが，注意しないと Fig. 6.8 のような状態になってしまいます．

入力時

(s e t q m e s s a g e (r e a d 3)) (c a r m e s s a g e) ……

ここまで読み込まれる

(read 3)の評価

(T h e c o n j u n c t i o n o f t h e m o o n s f i n d s I ……

(car message)はオーバーライトされ消失

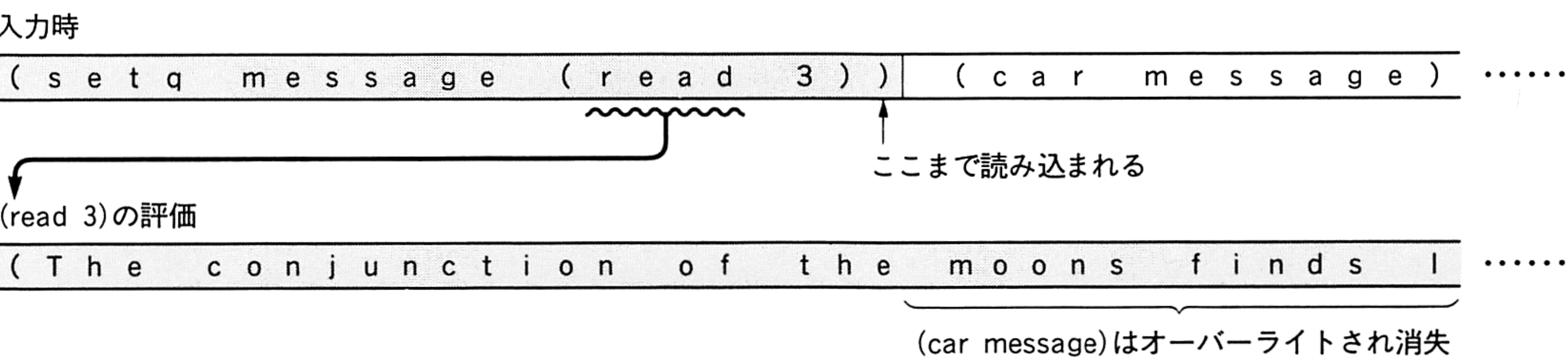



Fig. 6.8 入力バッファの破壊

Fig. 6.8 は *read* による入力文字列によって，未評価のＳ式が消されてしまったことを表しています．これを回避するために次のような関数を用意する必要があります．

● pushbuf()　(172 行～181 行)

Fig. 6.8 の誤動作の原因は入力先の変更に際しバッファの内容を保存しなかったことです．この関数はバッファの保存をする代わりに，未評価のＳ式にあたる文字列を ungetc() を用いてファイルに押し戻します．ただし，先頭の空白文字はいずれ読み飛ばされるものであるので，あえて押し戻す作業はしません．最後に読み込んだ文字から逆順に戻していくことに注意してください (Fig. 6.9)．

ここまで読み込み済み　ファイルに戻す

文字バッファ | c | a | l | f | o | ¥n | ¥0

③ ② ①

逆順に戻す

f | o | ¥n

ファイルバッファ

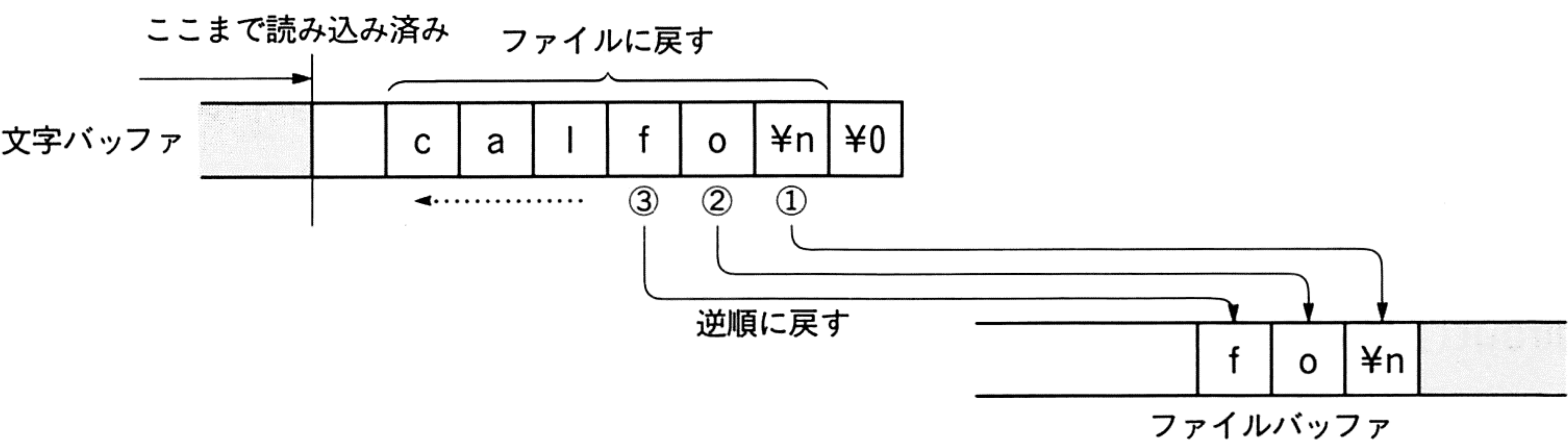



Fig. 6.9 ファイルの押し戻し

● readch_f()　(183 行～202 行)

readch は入力ファイルから 1 文字読み込み，シンボルアトムとして返します．入力先の変更にあたる部分は先の read_f() と大体同じですが，この関数では入力用の文字列バッファ oneline [] を使いませんので，バッファの押し戻し pushbuf() を呼んでいない点で異なります．入力が EOF だったら EOF を表すシンボルアトムへのポインタ eofread を返し，そうでなければその文字にあ

たる印字名を持つシンボルアトムがあるかどうかを old_atom()で検索し，すでに存在していれば そのシンボルアトムを，無ければ新しくシンボルアトムを登録して返します．

6.3.5 リダイレクト関数

特定のファイルに対し入出力をおこなう時は，いちいちファイルディスクリプタで入出力先を 指定するのは面倒ですので，デフォルトの入出力先を変更できると後が楽になります．

● dirin_f()　(204行〜212行)

関数 *dirin* は入力先を指定されたファイルディスクリプタで表されるファイルに向けます．入力 に関する変更をおこなうので，pushbuf()を呼ぶ必要があります．入力先を示すファイルポインタ cur_fpi を書き換えるのが主眼になります．

● dirout_f()　(214行〜222行)

dirout は出力先のファイルポインタ cur_fpo をファイルディスクリプタで指定されたファイル に変更します．

● dirin()　(224行〜239行)

dirin_f()の本体であり，また read_f()，readch_f()のための入力先変更にも用いられます．ファイルディスクリプタを car 部分に持つようなセルを引数とします．この引数が nil ならば，NULL を返して上位の関数に知らせ，その処理は上位関数にまかせます．

受け取ったファイルディスクリプタが適当な値ならば，入力先をそのファイルに向け，入力バッファをクリアします．

● dirout()　(241行〜255行)

dirout_f()の本体であり，print_f()などの出力関数のための出力先変更にも用いられます．基本 的な実行手順は，dirin()と同様です．

6.3.6 判別用の関数

次にあげる関数は FILE I/O に関する情報を提供します．

● fmode_f()　(257行〜274行)

fmode はファイルディスクリプタを引数にとり，そのファイルがどのようなモードでオープンさ

れているかを modestr [] に登録されている文字列を印字名に持つシンボルアトムを返します．ファイルがオープンされていない時は nil を返します．

```
% (fmode 0)
r                    ……ファイルディスクリプタ 0(標準入力)はテキストリードのモード
```

● **curin_f(), curout_f()** (276 行〜306 行)

curin, *curout* は現在の入力，出力がどのファイルに向いているかをファイルディスクリプタで知らせます．cur_fpi(cur_fpo) に等しいファイルポインタの，配列上での添字を数値アトムにして返します．

```
% (curin)
0                ……現在の入力先は標準入力
```

6.3.7 その他の入出力関数

● **load_f()** (308 行〜332 行)

関数 *load* は指定されたファイル(複数可)から S 式を読み込み，評価します．たとえば，

```
% (load 'kadorto 'katino)
```

と入力すれば 2 つのファイル kadorto.lsp と katino.lsp が読み込まれ，評価を受けます．内部では，読み込みと評価はトップレベルループ toplevel_f() を呼ぶことによって実現されます．for ループの中で，引数リストからのファイル名の取り出しと，オープン，読み込みと評価，クローズを，引数リストがなくなるまで繰り返しおこなっています．

● **prompt_f()** (334 行〜344 行)

プロンプトを変更しますが，引数として t が渡された場合には，標準プロンプト〝%〟にリセットします．引数が渡されなかった場合には，現在のプロンプトを返します．

● **format_f()** (346 行〜364 行)

関数 *format* は書式付き出力をおこなうための関数です．作用や引数については言語仕様を参照していただくことにして，ここではプログラム上の説明にとどめておきます．

関数 format_f() は引数のチェックと出力先の変更をおこない，*format* の実体は次の関数にあります．

● subform()　(366行～504行)

書式変換以外の部分は fputc()による1文字出力を使い，書式変換の部分は fprintf のフォーマットを活用します．tmp はこれから変換すべき引数リストの先頭を指しており，c は〝@〟オプションがついた時に ON となるフラグ，s は〝：〟オプションが付かず，通常の変換をする時に ON となるフラグです．文字ポインタ tps は変換文字列の中から書式変換の部分を抜き出す時に使います．

書式変換の始まりを表す〝~〟が表れるまで1文字出力をし，〝~〟が表れたら書式変換の部分の終わりまで tps を進めます．次に変換文字に応じて場合分けをし，書式変換の部分を fprintf()の仕様に合うように書き換えて fprintf()または fputc()で出力します．変換文字に〝：〟オプションが付いている時は出力する引数を次に進めずに，前回の出力に用いた引数をそのまま使います．

● copyform()　(506行～524行)

Lisp 上の変換文字列を fprintf()で使えるように変換します．〝s〟から〝e〟まで〝o〟で表される文字列に複写しますが，その際，〝%〟は〝%%〟に，〝~〟は〝%〟に，〝,〟は〝.〟に置き換え，〝：〟は無視します．こうして〝o〟で始まる文字列に fprintf()でそのまま使える書式文字列が作られます．

● ins_l()　(526行～546行)

先の copyform()と同様に書式文字列の複写をおこないますが，long の整数の書式変換に必要な〝l〟を変換文字の前に挿入するのが目的です．

List 6.19　lisp.h 追加

```
 1: typedef struct fstrct {
 2:     char    mode;
 3:     FILE    *ptr;
 4: } FILE_S;
 5:
 6: #define NFILES      10
 7:
 8: #define READFILE    2
 9: #define WRITEFILE   1
10: #define APPENDFILE  0
11: #define IFREAD      2
12: #define BINF        4
13:
14: #define FNA     34  /* File Not Available */
15: #define IFD     35  /* Illegal File Descripter */
16: #define FNO     36  /* File is not opend */
17: #define TMF     37  /* Too many Files */
18: #define FRO     38  /* File is Read Only */
19: #define FWO     39  /* File is Write Only */
20: #define CCF     40  /* Cannot Close File */
```

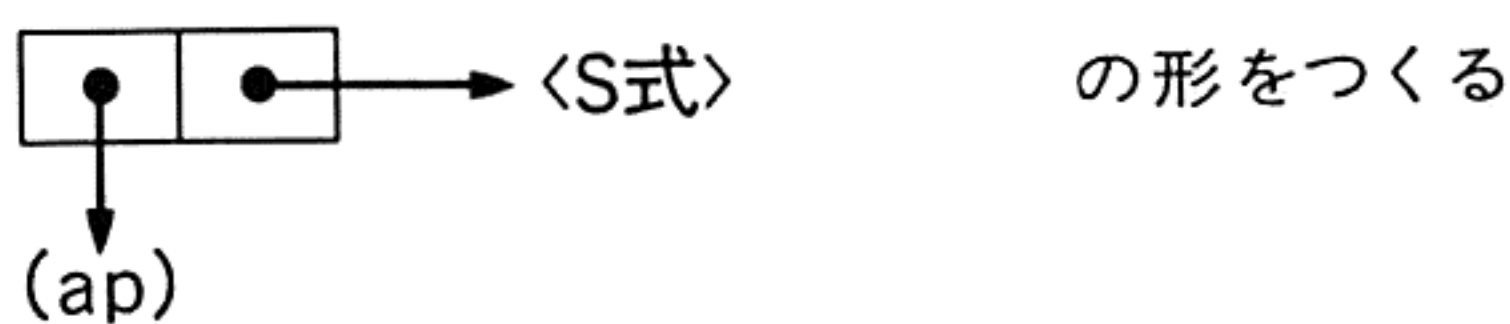
List 6.20 defvar.h 追加

```
 1: FILE_S  fp[NFILES];
 2: ATOMP   quote, stdprompt;
```

List 6.21 var.h 追加

```
 1: extern FILE_S   fp[NFILES];
 2: extern ATOMP    quote, stdprompt;
```

List 6.22 read.c escopt()を変更，formal()を追加

```
 1: static CELLP escopt(level)
 2: int level;
 3: {
 4:      CELLP   mk_list(), error();
 5:      CELLP   formal();
 6:      ATOMP   ret_atom();
 7:
 8:      switch(*txtp) {
 9:          case '(':
10:          case '[':   return mk_list(level);
11:          case '!':
12:          case '¥¥':  return (CELLP)ret_atom(ON);
13:          case '¥'':  return formal(level, quote); ………guoteの略記
14:          default:    return error(PSEXP);
15:      }
16: }
17:
18: static CELLP formal(level, ap)
19: int level;
20: ATOMP ap;                          [•|•]→<S式>       の形をつくる
21: {                                   ↓
22:      CELLP   cp2;                 (ap)
23:      CELLP   newcell();
24:
25:      stackcheck;
26:      *(++sp) = newcell();      ec;
27:      cp2 = newcell();          ec;
28:      (*sp)->car = (CELL *)ap;
29:      (*sp)->cdr = cp2;
30:      ++txtp;
31:      getcar(cp2, level);       ec;
32:      return *(sp--);
33: }
```

List 6.23 error.c err_msg[]へメッセージを追加

```
 1:      "File Not Available",
 2:      "Illegal File Descripter",
 3:      "File is not opened",
 4:      "Too many Files",
 5:      "File is Read Only",
 6:      "File is Write Only",
 7:      "Can't Close File",
```

List 6.24 main.c reset_err(), init(), mk_sys_atoms(), greeting()を変更

```
 1: static reset_err()
 2: {
 3:     cur_fpi = stdin;
 4:     cur_fpo = stdout;
 5:     err = NONERR;
 6:     txtp = oneline;
 7:     *txtp = '¥0';
 8:     for (sp = stacktop; sp < stacktop + STACKSIZ; ++sp)
 9:         *sp = (CELLP)nil;
10:     sp = stacktop - 1;
11:     verbos = ON;
12:     throwlabel = throwval = (CELLP)nil;
13:     prompt = stdprompt; ………プロンプトを元にもどす
14: }
15:
16: static init()
17: {
18:     char     *malloc();
19:     int      quit();
20:     int      i;
21:     CELLP    cp;
22:     ATOMP    ap;
23:     NUMP     np;
24:
25:     freecell = celltop = (CELLP)malloc(sizeof(CELL) * CELLSIZ);
26:     freeatom = atomtop = (ATOMP)malloc(sizeof(ATOM) * ATOMSIZ);
27:     freenum = numtop = (NUMP)malloc(sizeof(NUM) * NUMSIZ);
28:     newstr = strtop = (STR)malloc(STRSIZ);
29:     sp = stacktop = (CELLP *)malloc(sizeof(CELLP) * STACKSIZ);
30:
31:     if (freecell == NULL || freeatom == NULL
32:         || freenum == NULL || newstr == NULL || sp == NULL) {
33:         printf("Oops! Alloc Error : Too Large Data Area.¥n");
34:         printf("Please change --SIZ (defined in lisp.h).¥n");
35:         exit(1);
36:     }
37:
38:     nil = freeatom++;
39:
40:     for (cp = celltop; cp < celltop + CELLSIZ; ++cp) {
41:         cp->id = _CELL;
42:         cp->car = (CELLP)nil;
43:         cp->cdr = cp + 1;
44:     }
45:     (--cp)->cdr = (CELLP)nil;
46:
47:     for (ap = atomtop + 1; ap < atomtop + ATOMSIZ; ++ap) {
48:         ap->id = _ATOM;
49:         ap->plist = (CELLP)(ap + 1);
50:     }
51:     (--ap)->plist = (CELLP)nil;
52:
53:     for (np = numtop; np < numtop + NUMSIZ; ++np) {
54:         np->id = _FIX;
55:         np->value.ptr = np + 1;
56:     }
57:     (--np)->value.ptr = (NUMP)nil;
58:                                              ┐
59:     fp[0].mode = READFILE;                   │
60:     fp[0].ptr = stdin;                       │
61:     fp[1].mode = WRITEFILE;                  │
62:     fp[1].ptr = stdout;                      ├ ファイルディスクリプタの初期化
63:     fp[2].mode = WRITEFILE;                  │
64:     fp[2].ptr = stderr;                      │
65:     for (i = 3; i < NFILES; ++i)             │
66:         fp[i].ptr = NULL;                    │
67:                                              ┘
```

```
68:     for (i = 0; i < TABLESIZ; ++i)
69:         oblist[i] = (CELLP)nil;
70:
71:     mk_sys_atoms();
72:     ini_subr();
73:     signal( SIGINT, quit );
74: }
75:
76: static mk_sys_atoms()
77: {
78:     ATOMP   mk_atom();
79:     ATOMP   quote_f();
80:
81:     mk_nil();
82:     t = mk_atom("t");
83:     lambda = mk_atom("lambda");
84:     eofread = mk_atom("EOF");
85:     prompt = stdprompt = mk_atom("% ");
86:     quote = mk_atom("quote");                  ┐
87:     quote->ftype = _FSUBR;                     ├ quoteの定義はここでおこなってしまう
88:     quote->fptr = (CELLP)quote_f;              ┘
89: }
90:
91: static greeting()
92: {
93:     fprintf(stdout,"¥n");
94:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
95:     fprintf(stdout,"¥t           C A L F O¥n");
96:     fprintf(stdout,"¥t  Will o'Lisp Version 0.70¥n");
97:     fprintf(stdout,"¥t        (C)  1986, Mar¥n¥n");
98:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
99: }
```

List 6.25 inisubr.c ini_subr(), init0()を変更，init3()を追加

```
 1: ini_subr()
 2: {
 3:     init0();
 4:     init1();
 5:     init2();
 6:     init3();
 7: }
 8:
 9: static init0()
10: {
11:     CELLP car_f(), cdr_f(), cons_f();
12:     CELLP atom_f(), eq_f(), equal_f();
13:     CELLP de_f(), cond_f();                          ┐
14:     CELLP setq_f(), oblist_f(), quit_f();            │
15:     CELLP putprop_f(), get_f(), remprop_f();         │
16:     CELLP read_f(), terpri_f();                      │
17:     CELLP print_f(), prin1_f(), princ_f();           │
18:     CELLP minus_f(), plus_f();                       │
19:                                                      ├ quote_fの定義を削る
20:     defsubr("car",       car_f,       _SUBR);        │
21:     defsubr("cdr",       cdr_f,       _SUBR);        │
22:     defsubr("cons",      cons_f,      _SUBR);        │
23:     defsubr("atom",      atom_f,      _SUBR);        │
24:     defsubr("eq",        eq_f,        _SUBR);        │
25:     defsubr("equal",     equal_f,     _SUBR);        ┘
26:     defsubr("de",        de_f,        _FSUBR);
27:     defsubr("cond",      cond_f,      _FSUBR);
28:     defsubr("setq",      setq_f,      _FSUBR);
29:     defsubr("oblist",    oblist_f,    _SUBR);
30:     defsubr("quit",      quit_f,      _SUBR);
31:     defsubr("putprop",   putprop_f,   _SUBR);
```

```
32:     defsubr("get",       get_f,       _SUBR);
33:     defsubr("remprop",   remprop_f,   _SUBR);
34:     defsubr("read",      read_f,      _SUBR);
35:     defsubr("terpri",    terpri_f,    _SUBR);
36:     defsubr("print",     print_f,     _SUBR);
37:     defsubr("prinl",     prinl_f,     _SUBR);
38:     defsubr("princ",     princ_f,     _SUBR);
39:     defsubr("minus",     minus_f,     _SUBR);
40:     defsubr("plus",      plus_f,      _SUBR);
41: }
42:
43: static init3()
44: {
45:     CELLP format_f(), prompt_f(), load_f();
46:     CELLP open_f(), close_f(), fmode_f();
47:     CELLP readch_f(), dirin_f(), dirout_f();
48:     CELLP curin_f(), curout_f();
49:
50:     defsubr("format",    format_f,    _SUBR);
51:     defsubr("prompt",    prompt_f,    _SUBR);
52:     defsubr("load",      load_f,      _SUBR);
53:     defsubr("open",      open_f,      _SUBR);
54:     defsubr("close",     close_f,     _SUBR);
55:     defsubr("fmode",     fmode_f,     _SUBR);
56:     defsubr("readch",    readch_f,    _SUBR);
57:     defsubr("dirin",     dirin_f,     _SUBR);
58:     defsubr("dirout",    dirout_f,    _SUBR);
59:     defsubr("curin",     curin_f,     _SUBR);
60:     defsubr("curout",    curout_f,    _SUBR);
61: }
```

List 6.26 iofunc.c ファイルごと差し替え

```
 1: /*                                  */
 2: /*           IOFUNC                 */
 3: /*                                  */
 4: /*  input/output function           */
 5:
 6: #include     "lisp.h"
 7: #include     <ctype.h>
 8:
 9: #define      forever        for(;;)
10:
11: static STR modestr[] = {
12:     "a", "w", "r", "",
13:     "ab", "wb", "rb"
14:     };
15:
16: CELLP open_f(args) ………ファイルを開く
17: CELLP args;
18: {
19:     char     type, fname[NAMLEN];
20:     int      i;
21:     FILE     *lfp, *fopen();
22:     CELLP    error();
23:     NUMP     np, newnum();
24:     STR      mode;
25:
26:     if(args->id != _CELL)   return error(NEA);
27:     if(args->car->id != _ATOM)  return error(IAA);
28:     getfname(fname, (ATOMP)(args->car));
29:     if((args = args->cdr)->id != _CELL) mode = "r";
30:     else if(args->car->id != _ATOM) return error(IAA);
31:     else     mode = ((ATOMP)(args->car))->name;
32:     for(i = 3; i < NFILES; ++i)
33:         if(fp[i].ptr == NULL) {
34:             switch(*mode) {
```

```
 35:                 case 'w':   type = WRITEFILE;
 36:                             break;
 37:                 case 'a':   type = APPENDFILE;
 38:                             break;
 39:                 default:    type = READFILE;
 40:             }
 41:             if(*(mode+1) == 'b')     type |= BINF;
 42:             lfp = fopen(fname, modestr[type]);
 43:             if(lfp == NULL) return error(FNA);
 44:             fp[i].mode = type;
 45:             fp[i].ptr = lfp;
 46:             np = newnum();  ec;
 47:             np->value.fix = (long)i;
 48:             return (CELLP)np;
 49:         }
 50:     return error(TMF);
 51: }
 52:
 53: static getfname(o, ap) ………シンボルの印字名からファイル名を取り出す
 54: STR o;
 55: ATOMP ap;
 56: {
 57:     int i;
 58:     STR strcpy();
 59:     STR os = o;
 60:     STR s = ap->name;
 61:
 62:     for (i = 1; i <= NAMLEN-5; ++i)
 63:         *o++ = *s++;
 64:     *o = '¥0';
 65:     while (*os != '¥0')
 66:         if (*os++ == '.')   return;
 67:     strcpy(os, ".lsp");
 68: }
 69:
 70: CELLP close_f(arg) ………ファイルを閉じる
 71: CELLP arg;
 72: {
 73:     int      i;
 74:     FILE     *lfp;
 75:     CELLP    error();
 76:
 77:     if (arg->id != _CELL)         return error(NEA);
 78:     if (arg->car->id != _FIX)     return error(IFD);
 79:     i = (int)((NUMP)(arg->car))->value.fix;
 80:     if (i < 3 || i >= NFILES)     return error(IFD);
 81:     lfp = fp[i].ptr;
 82:     if (lfp == NULL)    return error(FNO);
 83:     if (cur_fpo == lfp) cur_fpo = stdout;
 84:     if (cur_fpi == lfp) {
 85:         *txtp = '¥0';
 86:         cur_fpi = stdin;
 87:     }
 88:     fp[i].ptr = NULL;
 89:     if (fclose(lfp))    return error(CCF);
 90:     return arg->car;
 91: }
 92:
 93: CELLP print_f(args)
 94: CELLP args;
 95: {
 96:     CELLP    result, prin();
 97:     FILE     *nfp = cur_fpo;
 98:
 99:     result = prin(args, ESCON); ec;
100:     fputc('¥n', cur_fpo);
101:     cur_fpo = nfp;
102:     return result;
103: }
```

```
104:
105: CELLP prinl_f(args)
106: CELLP args;
107: {
108:     CELLP   result, prin();
109:     FILE    *nfp = cur_fpo;
110:
111:     result = prin(args, ESCON);
112:     cur_fpo = nfp;
113:     return result;
114: }
115:
116: CELLP princ_f(args)
117: CELLP args;
118: {
119:     CELLP   result, prin();
120:     FILE    *nfp = cur_fpo;
121:
122:     result = prin(args, ESCOFF);
123:     cur_fpo = nfp;
124:     return result;
125: }
126:
127: static CELLP prin(args, mode) ………print/prinl/princで引数のチェックと
128: CELLP args;                            出力先の変更及び出力をおこなう
129: int mode;
130: {
131:     int     i;
132:     CELLP   error(), dirout();
133:
134:     if (args->id != _CELL)
135:         return error(NEA);
136:     dirout(args->cdr);  ec;
137:     print_s(args->car, mode);
138:     return args->car;
139: }
140:
141: CELLP terpri_f(arg)
142: CELLP arg;
143: {
144:     int     i;
145:     FILE    *nfp = cur_fpo;
146:     CELLP   error(), dirout();
147:
148:     dirout(arg);    ec;
149:     fputc('¥n', cur_fpo);
150:     cur_fpo = nfp;
151:     return (CELLP)nil;
152: }
153:
154: CELLP read_f(arg)
155: CELLP arg;
156: {
157:     int     i;
158:     FILE    *nfp = cur_fpi;
159:     CELLP   cp, error(), read_s(), dirin();
160:
161:     if (dirin(arg) != (CELLP)nil) {    ┐
162:         ec;                            │
163:         cp = read_s(TOP);              │
164:         pushbuf();                     ├ ファイルディスクリプタが指定された時
165:         cur_fpi = nfp;                 │
166:         return cp;                     ┘
167:     }
168:     cp = read_s(TOP);
169:     return cp;
170: }
```

```
171:
172: static pushbuf() ………入力バッファの退避
173: {
174:     STR tp;
175:
176:     while(isspace(*txtp)) ++txtp;
177:     for(tp = txtp; *tp != '¥0'; ++tp);
178:     for(--tp; tp >= txtp; --tp)
179:         ungetc(*tp, cur_fpi);
180:     *txtp = '¥0';
181: }
182:
183: CELLP readch_f(arg)
184: CELLP arg;
185: {
186:     char    ch[2];
187:     int     i;
188:     FILE    *nfp = cur_fpi;
189:     CELLP   error(), dirin();
190:     ATOMP   ap, old_atom(), mk_atom();
191:
192:     dirin(arg); ec;
193:     if(isatty(fileno(cur_fpi))) i = getch();
194:     else    i = fgetc(cur_fpi);
195:     if (i == EOF)   return (CELLP)eofread;
196:     ch[0] = i;
197:     ch[1] = '¥0';
198:     cur_fpi = nfp;
199:     if ((ap = old_atom(ch)) == NULL)
200:         return (CELLP)mk_atom(ch);
201:     return (CELLP)ap;
202: }
203:
204: CELLP dirin_f(arg)
205: CELLP arg;
206: {
207:     CELLP   cp, dirin();
208:
209:     if ((cp = dirin(arg)) == (CELLP)nil)
210:         return error(NEA);
211:     return cp;
212: }
213:
214: CELLP dirout_f(arg)
215: CELLP arg;
216: {
217:     CELLP   cp, dirout();
218:
219:     if ((cp = dirout(arg)) == (CELLP)nil)
220:         return error(NEA);
221:     return cp;
222: }
223:
224: CELLP dirin(arg) ………入力先の変更をおこなう
225: CELLP arg;
226: {
227:     int     i;
228:     CELLP   error();
229:
230:     if (arg->id != _CELL)       return (CELLP)nil;
231:     if (arg->car->id != _FIX)   return error(IFD);
232:     i = (int)((NUMP)(arg->car))->value.fix;
233:     if (i < 0 || i >= NFILES)   return error(IFD);
234:     if (fp[i].ptr == NULL)      return error(FNO);
235:     if (!(fp[i].mode & IFREAD)) return error(FWO);
236:     pushbuf();
237:     cur_fpi = fp[i].ptr;
238:     return arg->car;
239: }
```

```
240:
241: CELLP dirout(arg) ………出力先の変更をおこなう
242: CELLP arg;
243: {
244:     int      i;
245:     CELLP    error();
246:
247:     if (arg->id != _CELL)        return (CELLP)nil;
248:     if (arg->car->id != _FIX)    return error(IFD);
249:     i = (int)((NUMP)(arg->car))->value.fix;
250:     if (i < 0 || i >= NFILES)    return error(IFD);
251:     if (fp[i].ptr == NULL)       return error(FNO);
252:     if (fp[i].mode & IFREAD)     return error(FRO);
253:     cur_fpo = fp[i].ptr;
254:     return arg->car;
255: }
256:
257: CELLP fmode_f(arg) ………ファイルディスクリプタの状態を返す
258: CELLP arg;
259: {
260:     int      i;
261:     ATOMP    ap, old_atom(), mk_atom();
262:     CELLP    error();
263:     STR      s;
264:
265:     if (arg->id != _CELL)        return error(NEA);
266:     if (arg->car->id != _FIX)    return error(IFD);
267:     i = (int)((NUMP)(arg->car))->value.fix;
268:     if (i < 0 || i >= NFILES)    return error(IFD);
269:     if (fp[i].ptr == NULL)       return (CELLP)nil;
270:     s = modestr[fp[i].mode];
271:     if ((ap = old_atom(s)) == NULL)
272:         return (CELLP)mk_atom(s);
273:     return (CELLP)ap;
274: }
275:
276: CELLP curin_f(arg) ………現在の入力先ファイルのファイルディスクリプタを返す
277: CELLP arg;
278: {
279:     NUMP     np, newnum();
280:     int      i;
281:
282:     np = newnum();  ec;
283:     for (i = 0; i < NFILES; ++i) {
284:         if (cur_fpi == fp[i].ptr) {
285:             np->value.fix = (long)i;
286:             return (CELLP)np;
287:         }
288:     }
289:     return error(UNDEF);
290: }
291:
292: CELLP curout_f(arg) ………現在の出力先ファイルのファイルディスクリプタを返す
293: CELLP arg;
294: {
295:     NUMP     np, newnum();
296:     int      i;
297:
298:     np = newnum();  ec;
299:     for (i = 0; i < NFILES; ++i) {
300:         if (cur_fpo == fp[i].ptr) {
301:             np->value.fix = (long)i;
302:             return (CELLP)np;
303:         }
304:     }
305:     return error(UNDEF);
306: }
```

```
307:
308: CELLP load_f(arg)
309: CELLP arg;
310: {
311:     FILE    *lfp, *bak, *fopen();
312:     CELLP   error();
313:     char    fname[NAMLEN];
314:
315:     if (arg->id != _CELL)   return error(NEA);
316:     bak = cur_fpi;
317:     pushbuf();
318:     forever {
319:         if (arg->car->id != _ATOM)  return error(IAA);
320:         getfname(fname, (ATOMP)(arg->car));
321:         fprintf(cur_fpo, "¥nloading ... %s¥n", fname);
322:         if ((lfp = fopen(fname, "r")) == NULL)
323:                 return error(FNA);
324:         cur_fpi = lfp;
325:         toplevel_f(); ………トップレベルループを呼んでS式の読み込みと評価をおこなう
326:         fclose(lfp);     ec;
327:         *txtp = '¥0';
328:         if ((arg = arg->cdr)->id != _CELL)  break;
329:     }
330:     cur_fpi = bak;
331:     return (CELLP)t;
332: }
333:
334: CELLP prompt_f(arg)
335: CELLP arg;
336: {
337:     CELLP   error();
338:                                                  …引数がなければ現在
339:     if (arg->id != _CELL)   return (CELLP)prompt; …… のpromptを返す
340:     if ((arg = arg->car)->id != _ATOM)  return error(IAA);
341:     if ((ATOMP)arg == t)    prompt = stdprompt; ………引数がtならば標準が
342:     else prompt = (ATOMP)arg;                        promptに変更する
343:     return (CELLP)prompt;
344: }
345:
346: CELLP format_f(args) ………フォーマット文
347: CELLP args;
348: {
349:     int     i;
350:     FILE    *nfp = cur_fpo;
351:     CELLP   error(), dirout();
352:
353:     if (args->id != _CELL)  return error(NEA);
354:     if (args->car != (CELLP)nil)
355:         dirout(args);
356:     ec;
357:     if ((args = args->cdr)->id != _CELL)
358:         return error(NEA);
359:     if (args->car->id != _ATOM)
360:         return error(IAA);
361:     subform(((ATOMP)(args->car))->name, args->cdr);
362:     cur_fpo = nfp;
363:     return (CELLP)nil;
364: }
365:
366: static subform(tp, args)
367: STR tp;
368: CELLP args;
369: {
370:     CELLP   tmp = args;
371:     CELLP   error();
372:     STR     tps;
373:     char    c, s, fmt[NAMLEN];
374:
```

```
375:    while (*tp != '¥0') {
376:        for(;; ++tp) {
377:            if (*tp == '¥0') return;
378:            if (*tp == '~')  break;               "~"が出てくるまでは単に出力をするだけ
379:            if (iskanji(*tp))
380:                fputc(*tp++, cur_fpo);
381:            fputc(*tp, cur_fpo);
382:        }
383:        tps = tp;
384:        if (*(++tp) == '-')      ++tp;
385:        while (isdigit(*tp))     ++tp;
386:        if (*tp == ',')          ++tp;
387:        while (isdigit(*tp))     ++tp;
388:        if (*tp == ':') {
389:            ++tp;                                 各種指定のチェック
390:            s = 0;
391:        }
392:        else  s = 1;
393:        if (*tp == '@') {
394:            ++tp;
395:            c = 1;
396:        }
397:        else    c = 0;
398:        switch (*tp) {
399:            case 'd':
400:            case 'x':
401:            case 'o':
402:                if (s) {
403:                    tmp = args;                   継続オプションの無い時は引数をすすめる
404:                    args = args->cdr;
405:                }
406:                if (tmp->id != _CELL)
407:                    return (int)error(NEA);
408:                if (tmp->car->id != _FIX)
409:                    return (int)error(IAF);
410:                ins_l(fmt, tps, ++tp);
411:                fprintf(cur_fpo, fmt, ((NUMP)(tmp->car))->value.fix);
412:                break;
413:            case 'e':
414:            case 'f':
415:            case 'g':
416:                if (s) {
417:                    tmp = args;
418:                    args = args->cdr;
419:                }
420:                if (tmp->id != _CELL)
421:                    return (int)error(NEA);
422:                if (tmp->car->id != _FLT)
423:                    return (int)error(IAFL);
424:                copyform(fmt, tps, ++tp);
425:                fprintf(cur_fpo, fmt, ((NUMP)(tmp->car))->value.flt);
426:                break;
427:            case 'c':
428:                if (s) {
429:                    tmp = args;
430:                    args = args->cdr;
431:                }
432:                if (tmp->id != _CELL)
433:                    return (int)error(NEA);
434:                if (tmp->car->id != _FIX)
435:                    return (int)error(IAF);
436:                copyform(fmt, tps, ++tp);
437:                fprintf(cur_fpo, fmt, ((NUMP)(tmp->car))->value.fix);
438:                break;
439:            case 'a':
440:                if (s) {
441:                    tmp = args;
442:                    args = args->cdr;
```

```
443:                    }
444:                    if (tmp->id != _CELL)
445:                        return (int)error(NEA);
446:                    if (tmp->car->id != _ATOM) { ………引数がシンボルでない時は
447:                        print_s(tmp->car, ESCOFF);     princ出力をする
448:                        ++tp;
449:                    }
450:                    else { ………引数がシンボルの時は，引字名を"%S"出力する
451:                        copyform(fmt, tps, ++tp);
452:                        *(fmt + (tp - tps) - 1) = 's';
453:                        fprintf(cur_fpo, fmt, ((ATOMP)(tmp->car))->name);
454:                    }
455:                    break;
456:                case 's':
457:                    if (s) {
458:                        tmp = args;
459:                        args = args->cdr;
460:                    }
461:                    if (tmp->id != _CELL)
462:                        return (int)error(NEA);
463:                    print_s(tmp->car, ESCON);
464:                    ++tp;
465:                    break;
466:                case 'p':
467:                    if (s) {
468:                        tmp = args;
469:                        args = args->cdr;
470:                    }
471:                    ++tp;
472:                    if (tmp->id != _CELL)
473:                        return (int)error(NEA);
474:                    if (tmp->car->id != _FIX) break;
475:                    if (((NUMP)(tmp->car))->value.fix == 1) {
476:                        if (c)  fputc('y', cur_fpo);  |
477:                        break;                         | if(c)……
478:                    }                                  | "@"付きのpオプション
479:                    if (c)  fprintf(cur_fpo, "ies");   |
480:                    else    fputc('s', cur_fpo);
481:                    break;
482:                case 'n':
483:                    fputc('¥n', cur_fpo);
484:                    ++tp;
485:                    break;
486:                case 'r':
487:                    fputc('¥r', cur_fpo);
488:                    ++tp;
489:                    break;
490:                case 't':
491:                    fputc('¥t', cur_fpo);
492:                    ++tp;
493:                    break;
494:                case 'b':
495:                    fputc('¥b', cur_fpo);
496:                    ++tp;
497:                    break;
498:                default:                          |
499:                    args = tmp;                   | 適当な変換文字がなかったのでそのまま出力する
500:                    copyform(fmt, ++tps, ++tp);   |
501:                    fprintf(cur_fpo, fmt);        |
502:            }
503:        }
504: }
505:
506: static copyform(o, s, e)
507: STR o, s, e;
508: {
509:     while (s != e) {
510:         if (*s == ':') { ………":"は無視する
```

```
511:                ++s;
512:                continue;
513:            }
514:            if (*s == '%') {  ………"%"は"%%"に展開する
515:                *o++ = '%';
516:                *o = '%';
517:            }
518:            else if (*s == '~') *o = '%'; ……………………"~"は"%"に変換する
519:                else *o = (*s == ',' ? '.' : *s); ………", "は". "に変換する
520:            ++s;
521:            ++o;
522:        }
523:        *o = '¥0';
524: }
525:
526: static ins_l(o, s, e)
527: STR o, s, e;
528: {
529:        while (s != e - 1) {
530:            if (*s == ':')  {
531:                ++s;
532:                continue;
533:            }
534:            if (*s == '%') {
535:                *o++ = '%';
536:                *o = '%';
537:            }
538:            else if (*s == '~') *o = '%';
539:                else *o = (*s == ',' ? '.' : *s);
540:            ++s;
541:            ++o;
542:        }
543:        *o++ = 'l'; ………変換文字の前に"l"を入れる
544:        *o++ = *s;  ………変換文字をコピーする
545:        *o = '¥0';
546: }
```

6.4 機能拡張(4) Lispの常備関数の作成 ―MONTINOバージョン―

この段階では，一挙に関数の数を増加して，日常生活に不便のないようにします．たとえば，ちょっとリストが作りたいな，という時に，いちいち *cons* を使ってセルをつなぐ代わりに，*list* を使って一発で任意の長さのリストを作る，という文化的生活が送れるようになるわけです．もっともリストを作るのに *list* を使っているようでは〝現代的〟とはいい難く，せいぜい昭和30年ころの〝文化的〟という言葉がふさわしいところです．リストを作るには今は〝バッククォート〟を使います．ともあれ，この拡張によって追加される数多くの関数を使って，余裕あるプログラミングを楽しんでください．なお，各関数の機能の詳細については，Will o'Lisp のマニュアルを見てください．

6.4.1 リスト処理用関数の追加 (List 6.31)

fun.c には，リスト処理用の関数が追加されます．いずれも LISP1.5 以来の伝統的なものです．

● append_f() (1行～52行)

関数 *append* の定義です．*append* は，いくつかのリストを引数にとり，それらをつないだリストを返します．引数として与えられたリストが変更を受けないように，返されたリストは引数の複製をつないだものになっています．

2つのリストを接続するには，前方に置かれるリストの最後のセルの cdr を書き換えて，後ろのリストの先頭を指すようにします．したがって書き換えられるのは前のリストだけです．3つ以上のリストを *append* する時も，最後のリストは変更を受けないので，複製を作る必要はありません．

一般に，リストを返す関数では，リストの要素がひとつもないことが判明した時には，nil を返さなければなりません．この時はセルはひとつも用意する必要がありません．*append* の場合，すべての引数が nil ならば，nil を返すことになります．したがって，引数を順番に調べ，nil でない要素が見つかってから，セルを用意するようにしています．また，リストを返す関数では，返すリストの先頭を押さえておく必要があります．(〝リストを返す〟とはリストの先頭へのポインタを返すことだからです．)これは，sp に置かなければなりません．ローカル変数に置くと，ガベージコレクタからそのリストが見えないため，*append* の実行中にガベージコレクトが起きた時に，そのリストを構成しているセルが回収されてしまうからです．

● nconc_f() (54行〜82行)

nconc は，*append* のようにリストをコピーせず，そのままポインタを書き換えてつないでしまう関数です．その点を除けばほぼ *append* と同じ作りです．

● reverse_f() (84行〜98行)

reverse は，与えられたリストの要素を逆順に並べたリストを作って返す関数です．要素を前から順に取り出して，*cons* するだけで実現しています．

● list_f() (100行〜119行)

list は与えられた引数をすべてひとつのリストにまとめて返します．

● assoc_f() (121行〜132行)

assoc は連想リスト(association list)を検索する関数です．連想リストとは，次のようなものです．

((Sunday . 日)(Monday . 月)(Tuesday . 火))

連想リストの要素はドット対であり，各ドット対の car にはあるキーワードが，cdr にはそのキーワードに関連する(associate：連想される)データが置かれます．このように，連想リストは一種の辞書，あるいは変換テーブルとして使用します．*assoc* は，このリストとキーワードを引数に与えると，そのキーワードを含むドット対を返す関数です．見つからなければ，nil を返します．

```
% (setq dictionary '((Sunday . 日)(Monday . 月)(Tuesday . 火)))
((Sunday . 日)(Monday . 月)(Tuesday . 火))

% (assoc 'Monday dictionary)
(Monday . 月)

% (assoc 'Wednesday dictionary)
nil
```

この機能はすでに関数 *prog* を実現するのに利用されています．C プログラム中から呼べるように，実際の仕事は prog.c(List 6.18)に含まれる assoc()がおこない，assoc_f()はこれを呼ぶだけになっています．

● rplaca_f() (134行〜145行)

引数の car を書き換える関数です.

● rplacd_f() (147行〜158行)

引数の cdr を書き換える関数です.

● length_f() (160行〜175行)

リストに含まれる要素の個数を返す関数です.

6.4.2 特殊形式の追加 (List 6.32)

control.c には, 制御構造としての and, or および setq の変形が加えられます.

● and_f() (1行〜14行)

原則として引数を順番に評価し, t を返します. 一方, 評価結果が nil であるものを発見すると, 残りの引数の評価はせずにその時点で実行を終了し, nil を返します.

● or_f() (16行〜28行)

原則として引数を順番に評価し, nil を返します. 一方, 評価結果が nil でないものを発見すると, 残りの引数の評価はせずにその時点で実行を終了し, t を返します.

● psetq_f() (30行〜61行)

setq と同様ですが, いくつかの代入を指定した時, *psetq* は, それらが並行しておこなわれる所が異なっています. 並行して代入がおこなわれるとは, 代入によって変化した環境が, 次の代入に影響を及ぼさないことを意味します. したがって, すべての引数の評価を *psetq* に入る前の環境でおこなった後, 代入を実施するようにしています.

● set_f() (63行〜85行)

setq と同様に代入をおこないますが, *setq* では変数が評価されなかったのに対し, 変数部分も評価がおこなわれるため, 実行時に代入先を動的に決めることができます.

6.4.3 算術関数の追加 (List 6.33)

calc.cには，乗除算を含む，いくつかの算術関数を追加します．

● add1_f()，sub1_f() (1行～40行)

それぞれ，(plus x 1)，(difference x 1)と同義です．

● times_f()，difference_f()，quotient_f()，remainder_f() (42行～145行)

積，差，商，剰余を求めます．plus_f()の演算記号〝+〞を〝*〞，〝-〞，〝/〞，〝%〞に換えただけの関数です．ただし，remainder_f()は浮動小数点型を受け付けません．

● divide_f() (147行～165行)

商と剰余をリストにして返します．新しいリストを作る関数ですので，スタックにリストの先頭を退避する処理が必要になります．

6.4.4 述語関数の追加 (List 6.34)

拡張に伴い，いくつかの述語関数を追加します．これらはpred.cファイルにまとめられます．

● null_f()，zerop_f()，numberp_f()，greaterp_f()，lessp_f() (10行～114行)

これらはリストを見てもらえば，すぐにわかるでしょう．greaterp_f()やlessp_f()は算述関数のplus_f()などとまったく同じ構造を持っています．

● member_f() (116行～131行)

第1引数が，第2引数のリストの中にあれば，その要素から後を返します．たとえば，

```
% (member 'Znd '(VAN pin Znd MEC))
(Znd MEC)
```

となります．この関数はequal_f()の下位関数equal()を呼び出しています．

6.4.5 文字列操作関数の追加 (List 6.35)

文字列というデータ型はWill o'Lispにはありません．したがって，ここでいうところの文字列とはシンボルアトムの印字名のことです．str.cファイルとして追加される関数群はシンボルアトムの印字名に対して操作を加えます．また，そのため，read.c内にあるシンボルアトム管理の関数を下位関数として呼び出していますので，そちらも参照してください．

● ascii_f() (11行〜34行)

シンボルアトムをアスキーコードに，アスキーコードをシンボルアトムに双方向の変換をする関数です．シンボルアトムの印字名は先頭の1バイトのみを変換します．すなわち，漢字コードに関する処理については考慮されていません．

● implode_f() (36行〜58行)

シンボルアトムを要素とするリストを引数にとり，その要素のシンボルアトムのすべての印字名をつなげた文字列を印字名とするシンボルアトムを作り出します．たとえば，

```
% (implode '(L o r d))
Lord
```

となります．新しいシンボルアトムの印字名用のバッファに下位関数strapnd()を用いて各印字名を複写し，その印字名を持つシンボルアトムを返します．

● explode_f() (60行〜98行)

シンボルアトムの印字名を1文字毎に分解しリストの形にして返します．ここであげた関数群の中で，この関数だけは漢字に対する考慮をしてあります．漢字を印字名に持つシンボルアトムを分解した時に見苦しくなってしまうからです．また，この関数は数値を文字列として分解する機能もあります．この例は，

```
% (explode 123)
(¥1 ¥2 ¥3)
```

のようになります．この数値の分解ではsprintf()を用いています．

● alen_f() (100行〜111行)

シンボルアトムの印字名の長さをバイト単位で返します．もちろん文字列の最後のヌル文字は含みません．

List 6.27 lisp.h 追加

```
1: #define DIVZERO 41  /* Division by zero */
```

List 6.28 error.c err_msg[]ヘメッセージを追加

```
1:     "Division by Zero",
```

List 6.29 main.c greeting()を変更

```
1: static greeting()
2: {
3:     fprintf(stdout,"¥n");
4:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
5:     fprintf(stdout,"¥t          M O N T I N O¥n");
6:     fprintf(stdout,"¥t  Will o'Lisp Version 0.80¥n");
7:     fprintf(stdout,"¥t         (C)  1986, Mar¥n¥n");
8:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
9: }
```

List 6.30 inisubr.c ini_subr()を変更, init4()を追加

```
 1: ini_subr()
 2: {
 3:     init0();
 4:     init1();
 5:     init2();
 6:     init3();
 7:     init4();
 8: }
 9:
10: static init4()
11: {
12:     CELLP and_f(), or_f(), psetq_f(), set_f();
13:     CELLP append_f(), nconc_f(), list_f(), reverse_f();
14:     CELLP assoc_f(), rplaca_f(), rplacd_f(), length_f();
15:     CELLP member_f(), null_f(), zerop_f(), numberp_f();
16:     CELLP lessp_f(), greaterp_f();
17:     CELLP ascii_f(), implode_f(), explode_f(), alen_f();
18:     CELLP add1_f(), sub1_f(), difference_f(), times_f();
19:     CELLP remainder_f(), quotient_f(), divide_f();
20:
21:     defsubr("and",      and_f,      _FSUBR);
22:     defsubr("or",       or_f,       _FSUBR);
23:     defsubr("psetq",    psetq_f,    _FSUBR);
24:     defsubr("set",      set_f,      _FSUBR);
25:     defsubr("append",   append_f,   _SUBR);
26:     defsubr("nconc",    nconc_f,    _SUBR);
27:     defsubr("list",     list_f,     _SUBR);
28:     defsubr("reverse",  reverse_f,  _SUBR);
29:     defsubr("assoc",    assoc_f,    _SUBR);
30:     defsubr("rplaca",   rplaca_f,   _SUBR);
31:     defsubr("rplacd",   rplacd_f,   _SUBR);
32:     defsubr("member",   member_f,   _SUBR);
33:     defsubr("null",     null_f,     _SUBR);
34:     defsubr("zerop",    zerop_f,    _SUBR);
35:     defsubr("numberp",  numberp_f,  _SUBR);
36:     defsubr("lessp",    lessp_f,    _SUBR);
37:     defsubr("greaterp", greaterp_f, _SUBR);
38:     defsubr("length",   length_f,   _SUBR);
39:     defsubr("ascii",    ascii_f,    _SUBR);
```

```
40:     defsubr("implode",  implode_f,   _SUBR);
41:     defsubr("explode",  explode_f,   _SUBR);
42:     defsubr("alen",     alen_f,      _SUBR);
43:     defsubr("add1",     add1_f,      _SUBR);
44:     defsubr("sub1",     sub1_f,      _SUBR);
45:     defsubr("difference",   difference_f,    _SUBR);
46:     defsubr("times",    times_f,     _SUBR);
47:     defsubr("quotient", quotient_f, _SUBR);
48:     defsubr("remainder",    remainder_f,     _SUBR);
49:     defsubr("divide",   divide_f,    _SUBR);
50: }
```

List 6.31 fun.c 各種のリスト処理関数を追加

```
 1: CELLP append_f(args)
 2: CELLP args;
 3: {
 4:     CELLP   cp1, cp2, newcell(), error();
 5:
 6:     if (args->id != _CELL)
 7:         return error(NEA);
 8:     while (args->car->id != _CELL) {        ┐
 9:         if (args->car != (CELLP)nil)        │
10:             return error(IAL);              │
11:         if (args->cdr->id != _CELL)         │ nilは無視する
12:             return args->car;               │
13:         args = args->cdr;                   ┘
14:     }
15:     if (args->cdr->id != _CELL)             ┐ 最後の引数ならそのまま返す
16:         return args->car;                   ┘
17:     *++sp = newcell(); ec;                  ┐
18:     cp1 = *sp;                              │
19:     cp2 = args->car;                        │
20:     cp1->car = cp2->car;                    │
21:     cp2 = cp2->cdr;                         │
22:     while (cp2->id == _CELL) {              │ 前につなぐリストはコピーをとる
23:         cp1->cdr = newcell(); ec;           │
24:         cp1 = cp1->cdr;                     │
25:         cp1->car = cp2->car;                │
26:         cp2 = cp2->cdr;                     │
27:     }                                       ┘
28:     args = args->cdr;
29:     while (args->cdr->id == _CELL) {
30:         while (args->car->id != _CELL) {
31:             if (args->car != (CELLP)nil)
32:                 return error(IAL);
33:             if (args->cdr->id != _CELL) {
34:                 cp1->cdr = (CELLP)nil;
35:                 return *sp--;
36:             }
37:             args = args->cdr;
38:         }
39:         cp2 = args->car;
40:         while (cp2->id == _CELL) {
41:             cp1->cdr = newcell(); ec;
42:             cp1 = cp1->cdr;
43:             cp1->car = cp2->car;
44:             cp2 = cp2->cdr;
45:         }
46:         args = args->cdr;
47:     }
48:     if (args->car->id != _CELL && args->car != (CELLP)nil)
49:         return error(IAL);
50:     cp1->cdr = args->car;
51:     return *sp--;
52: }
```

```
 53:
 54: CELLP nconc_f(args)
 55: CELLP args;
 56: {
 57:     CELLP   cp1, cp2, error();
 58:
 59:     if (args->id != _CELL)
 60:         return error(NEA);
 61:     while (args->car == (CELLP)nil && args->cdr->id == _CELL)………nilは無視する
 62:         args = args->cdr;
 63:     if (args->car->id != _CELL && args->car != (CELLP)nil)
 64:         return error(IAL);
 65:     if (args->cdr->id != _CELL)
 66:         return args->car;
 67:     cp1 = args->car;
 68:     cp2 = cp1;
 69:     args = args->cdr;
 70:     while (cp2->cdr->id == _CELL)     |
 71:         cp2 = cp2->cdr;               } 最初のリストの最後のセルを取り出し，cdrを書きかえる
 72:     cp2->cdr = args->car;             |
 73:     while (args->cdr->id == _CELL) {
 74:         args = args->cdr;
 75:         if (args->car->id != _CELL && args->car != (CELLP)nil)
 76:             return error(IAL);
 77:         while (cp2->cdr->id == _CELL)
 78:             cp2 = cp2->cdr;
 79:         cp2->cdr = args->car;
 80:     }
 81:     return cp1;
 82: }
 83:
 84: CELLP reverse_f(args) ………リストを逆順にする
 85: CELLP args;
 86: {
 87:     CELLP   cp, cons();
 88:
 89:     if (args->id != _CELL)
 90:         return error(NEA);
 91:     cp = args->car;
 92:     *++sp = (CELLP)nil;
 93:     while (cp->id == _CELL) {
 94:         *sp = cons(cp->car, *sp);
 95:         cp = cp->cdr;
 96:     }
 97:     return *sp--;
 98: }
 99:
100: CELLP list_f(args) ………リストを作る
101: CELLP args;
102: {
103:     CELLP   cp, newcell();
104:
105:     if (args->id != _CELL)
106:         return (CELLP)nil;
107:     *++sp = newcell(); ec;
108:     cp = *sp;
109:     cp->car = args->car;
110:     args = args->cdr;
111:     while (args->id == _CELL) {
112:         cp->cdr = newcell(); ec;
113:         cp = cp->cdr;
114:         cp->car = args->car;
115:         args = args->cdr;
116:     }
117:     cp->cdr = (CELLP)nil;
118:     return *sp--;
119: }
```

```
120:
121: CELLP assoc_f(args) ………連想リストを検索する
122: CELLP args;
123: {
124:     CELLP   result, assoc();
125:
126:     if (args->id != _CELL || args->cdr->id != _CELL)
127:         return error(NEA);
128:     result = assoc(args->car, args->cdr->car); ec;
129:     if (!result)
130:         return (CELLP)nil;
131:     return result;
132: }
133:
134: CELLP rplaca_f(args) ………リストのcarを書き換える
135: CELLP args;
136: {
137:     CELLP   cp, error();
138:
139:     if (args->id != _CELL || args->cdr->id != _CELL)
140:         return error(NEA);
141:     if ((cp = args->car)->id != _CELL)
142:         return error(IAL);
143:     cp->car = args->cdr->car;
144:     return cp;
145: }
146:
147: CELLP rplacd_f(args) ………リストのcdrを書き換える
148: CELLP args;
149: {
150:     CELLP   cp, error();
151:
152:     if (args->id != _CELL || args->cdr->id != _CELL)
153:         return error(NEA);
154:     if ((cp = args->car)->id != _CELL)
155:         return error(IAL);
156:     cp->cdr = args->cdr->car;
157:     return cp;
158: }
159:
160: CELLP length_f(args) ………リストの長さを求める
161: CELLP args;
162: {
163:     long    i = 0;
164:     CELLP   error();
165:     NUMP    np, newnum();
166:
167:     if (args->id != _CELL)  return error(NEA);
168:     if (args->car->id != _CELL && args->car != (CELLP)nil)
169:         return error(IAL);
170:     for(args = args->car; args->id == _CELL; args = args->cdr)
171:         ++i;
172:     np = newnum();  ec;
173:     np->value.fix = i;
174:     return (CELLP)np;
175: }
```

List 6.32 control.c and_f(), or_f(), psetq_f(), set_f()を追加

```
 1: CELLP and_f(args, env)
 2: CELLP args, env;
 3: {
 4:     CELLP   result, eval();
 5:
 6:     result = (CELLP)t;
 7:     while (args->id == _CELL) {
 8:         result = eval(args->car, env); ec;
 9:         if (result == (CELLP)nil)
10:             return (CELLP)nil;
11:         args = args->cdr;
12:     }
13:     return result;
14: }
15:
16: CELLP or_f(args, env)
17: CELLP args, env;
18: {
19:     CELLP   result, eval();
20:
21:     while (args->id == _CELL) {
22:         result = eval(args->car, env); ec;
23:         if (result != (CELLP)nil)
24:             return result;
25:         args = args->cdr;
26:     }
27:     return (CELLP)nil;
28: }
29:
30: CELLP psetq_f(args, env)
31: CELLP args, env;
32: {
33:     CELLP   val, cp, *cur_sp, *sp2;
34:     CELLP   setenv(), eval(), error();
35:     ATOMP   var;
36:
37:     cur_sp = sp;
38:     for (cp = args; cp->id == _CELL; cp = cp->cdr->cdr) {
39:         if (cp->cdr->id != _CELL)
40:             return error(NEA);
41:         stackcheck;
42:         *++sp = eval(cp->cdr->car, env); ec;
43:     }
44:     val = *sp;
45:     sp2 = cur_sp+1;
46:     while(args->id == _CELL) {
47:         var = (ATOMP)args->car;
48:         if (var->id != _ATOM)
49:             return error(IAA);
50:         if (var == nil || var == t || var == eofread)
51:             return error(CCC);
52:         cp = setenv(var, *sp2, env); ec;
53:         if (cp == NULL) {
54:             var->value = *sp2;
55:         }
56:         sp2++;
57:         args = args->cdr->cdr;
58:     }
59:     sp = cur_sp;
60:     return val;
61: }
62:
63: CELLP set_f(args, env)
64: CELLP args, env;
65: {
66:     CELLP   val, result, setenv(), eval(), error();
```

(7〜11行) nilが出てくるまでbodyを順に評価する

(21〜25行) nilでないものが出てくるまでbodyを順に評価する

(38〜43行) setする値をすべて先に評価し，stackに積む

(50〜51行) 値を変えてはいけないsymbol御三家

```
67:     ATOMP   var;
68:
69:     while(args->id == _CELL) {
70:         if (args->cdr->id != _CELL)
71:             return error(NEA);
72:         var = (ATOMP)eval(args->car, env); ec; ………var を評価してとり出す点が
73:         val = eval(args->cdr->car, env); ec;          setq と異なる
74:         if (var->id != _ATOM)
75:             return error(IAA);
76:         if (var == nil || var == t || var == eofread)
77:             return error(CCC);
78:         result = setenv(var, val, env); ec;
79:         if (result == NULL) {
80:             var->value = val;
81:         }
82:         args = args->cdr->cdr;
83:     }
84:     return val;
85: }
```

List 6.33 calc.c 各種の算術関数を追加

```
 1: CELLP add1_f(arg)
 2: CELLP arg;
 3: {
 4:     char    c;
 5:     NUMP    np, newnum();
 6:     CELLP   error();
 7:
 8:     if (arg->id != _CELL)
 9:         return error(NEA);
10:     if ((c = arg->car->id) != _FIX && c != _FLT)
11:         return error(IAN);
12:     np = newnum();  ec;
13:     if (c == _FIX)
14:         np->value.fix = ((NUMP)(arg->car))->value.fix + 1;
15:     else {
16:         np->id = _FLT;
17:         np->value.flt = ((NUMP)(arg->car))->value.flt + 1;
18:     }
19:     return (CELLP)np;
20: }
21:
22: CELLP sub1_f(arg)
23: CELLP arg;
24: {
25:     char    c;
26:     NUMP    np, newnum();
27:
28:     if (arg->id != _CELL)
29:         return error(NEA);
30:     if ((c = arg->car->id) != _FIX && c != _FLT)
31:         return error(IAN);
32:     np = newnum();  ec;
33:     if (c == _FIX)
34:         np->value.fix = ((NUMP)(arg->car))->value.fix - 1;
35:     else {
36:         np->id = _FLT;
37:         np->value.flt = ((NUMP)(arg->car))->value.flt - 1;
38:     }
39:     return (CELLP)np;
40: }
41:
42: CELLP times_f(args)
43: CELLP args;
```

```
 44: {
 45:     CELLP   setfirst(), getlarg();
 46:     NUM     val;
 47:     NUMP    np;
 48: 
 49:     args = setfirst(args, &np); ec;
 50:     if (args == (CELLP)nil)     return (CELLP)np;
 51:     args = getlarg(args, &val); ec;
 52:     if (np->id == _FIX) {
 53:         while(val.id == _FIX) {
 54:             np->value.fix *= val.value.fix;
 55:             if (args == (CELLP)nil) return (CELLP)np;
 56:             args = getlarg(args, &val); ec;
 57:         }
 58:         toflt(np);
 59:     }
 60:     else    toflt(&val);
 61:     forever {
 62:         np->value.flt *= val.value.flt;
 63:         if (args == (CELLP)nil) return (CELLP)np;
 64:         args = getlarg(args, &val); ec;
 65:         toflt(&val);
 66:     }
 67: }
 68: 
 69: CELLP difference_f(args)
 70: CELLP args;
 71: {
 72:     CELLP   setfirst(), getlarg();
 73:     NUM     val;
 74:     NUMP    np;
 75: 
 76:     args = setfirst(args, &np); ec;
 77:     if (args == (CELLP)nil)     return (CELLP)np;
 78:     args = getlarg(args, &val); ec;
 79:     if (np->id == _FIX) {
 80:         while(val.id == _FIX) {
 81:             np->value.fix -= val.value.fix;
 82:             if (args == (CELLP)nil) return (CELLP)np;
 83:             args = getlarg(args, &val); ec;
 84:         }
 85:         toflt(np);
 86:     }
 87:     else    toflt(&val);
 88:     forever {
 89:         np->value.flt -= val.value.flt;
 90:         if (args == (CELLP)nil) return (CELLP)np;
 91:         args = getlarg(args, &val); ec;
 92:         toflt(&val);
 93:     }
 94: }
 95: 
 96: CELLP quotient_f(args)
 97: CELLP args;
 98: {
 99:     CELLP   setfirst(), getlarg(), error();
100:     NUM     val;
101:     NUMP    np;
102: 
103:     args = setfirst(args, &np); ec;
104:     if (args == (CELLP)nil)     return (CELLP)np;
105:     args = getlarg(args, &val); ec;
106:     if (np->id == _FIX) {
107:         while(val.id == _FIX) {
108:             if (val.value.fix == 0)
109:                 return error(DIVZERO);
110:             np->value.fix /= val.value.fix;
```

```
111:             if (args == (CELLP)nil) return (CELLP)np;
112:             args = getlarg(args, &val); ec;
113:         }
114:         toflt(np);
115:     }
116:     else    toflt(&val);
117:     forever {
118:         if (val.value.flt == 0)
119:             return error(DIVZERO);
120:         np->value.flt /= val.value.flt;
121:         if (args == (CELLP)nil) return (CELLP)np;
122:         args = getlarg(args, &val); ec;
123:         toflt(&val);
124:     }
125: }
126:
127: CELLP remainder_f(args)
128: CELLP args;
129: {
130:     CELLP   setfirst(), getlarg();
131:     NUM     val;
132:     NUMP    np;
133:
134:     args = setfirst(args, &np); ec;
135:     if (args == (CELLP)nil)     return (CELLP)np;
136:     args = getlarg(args, &val); ec;
137:     if (np->id == _FIX) {
138:         while(val.id == _FIX) {
139:             np->value.fix %= val.value.fix;
140:             if (args == (CELLP)nil) return (CELLP)np;
141:             args = getlarg(args, &val); ec;
142:         }
143:     }
144:     return error(IAF);
145: }
146:
147: CELLP divide_f(args)
148: CELLP args;
149: {
150:     CELLP   quotient_f(), remainder_f();
151:     CELLP   cp, newcell(), error();
152:
153:     stackcheck;
154:     *(++sp) = quotient_f(args); ec;
155:     stackcheck;
156:     *(++sp) = remainder_f(args); ec;
157:     stackcheck;
158:     *(++sp) = newcell();    ec;
159:     cp = newcell(); ec;
160:     cp->car = *(sp - 2);
161:     cp->cdr = *sp;
162:     (*sp)->car = *(sp - 1);
163:     sp -= 3;
164:     return cp;
165: }
```

List 6.34 pred.c 新しいファイルを作成

```
 1: /*                                    */
 2: /*           PRED                     */
 3: /*                                    */
 4: /*       predicates of lisp           */
 5: /*                                    */
 6:
 7: #include     "lisp.h"
 8: #define      forever      for(;;)
 9:
10: CELLP null_f(args)
11: CELLP args;
12: {
13:     CELLP   error();
14:
15:     if (args->id != _CELL)
16:         return error(NEA);
17:     if (args->car == (CELLP)nil)
18:         return (CELLP)t;
19:     return (CELLP)nil;
20: }
21:
22: CELLP zerop_f(arg)
23: CELLP arg;
24: {
25:     char    c;
26:     CELLP   error();
27:
28:     if (arg->id != _CELL)   return error(NEA);
29:     if ((c = arg->car->id) != _FIX && c != _FLT)
30:         return error(IAN);
31:     if (c == _FIX) {
32:         if (((NUMP)(arg->car))->value.fix == 0)
33:             return (CELLP)t;
34:     }
35:     else if (((NUMP)(arg->car))->value.flt == 0)
36:         return (CELLP)t;
37:     return (CELLP)nil;
38: }
39:
40: CELLP numberp_f(arg)
41: CELLP arg;
42: {
43:     char    c;
44:     CELLP   error();
45:
46:     if (arg->id != _CELL)   return error(NEA);
47:     if ((c = arg->car->id) != _FIX && c != _FLT)
48:         return (CELLP)nil;
49:     return (CELLP)t;
50: }
51:
52: CELLP greaterp_f(args) ………plusのROUTINEと同じアルゴリズムを使っている
53: CELLP args;
54: {
55:     CELLP   getlarg(), error();
56:     NUM     val1, val2;
57:
58:     args = getlarg(args, &val1);    ec;
59:     if (args == (CELLP)nil)     return error(NEA);
60:     args = getlarg(args, &val2);    ec;
61:     if (val1.id == _FIX) {
62:         while(val2.id == _FIX) {
63:             if (val1.value.fix <= val2.value.fix)
64:                 return (CELLP)nil;
65:             if (args == (CELLP)nil)
66:                 return (CELLP)t;
```

```
 67:             val1.value.fix = val2.value.fix;
 68:             args = getlarg(args, &val2);    ec;
 69:         }
 70:         toflt(&val1);
 71:     }
 72:     else toflt(&val2);
 73:     forever {
 74:         if (val1.value.flt <= val2.value.flt)
 75:             return (CELLP)nil;
 76:         if (args == (CELLP)nil)
 77:             return (CELLP)t;
 78:         val1.value.fix = val2.value.fix;
 79:         args = getlarg(args, &val2);    ec;
 80:         toflt(&val2);
 81:     }
 82: }
 83:
 84: CELLP lessp_f(args)
 85: CELLP args;
 86: {
 87:     CELLP   getlarg(), error();
 88:     NUM     val1, val2;
 89:
 90:     args = getlarg(args, &val1);    ec;
 91:     if (args == (CELLP)nil)     return error(NEA);
 92:     args = getlarg(args, &val2);    ec;
 93:     if (val1.id == _FIX) {
 94:         while(val2.id == _FIX) {
 95:             if (val1.value.fix >= val2.value.fix)
 96:                 return (CELLP)nil;
 97:             if (args == (CELLP)nil)
 98:                 return (CELLP)t;
 99:             val1.value.fix = val2.value.fix;
100:             args = getlarg(args, &val2);    ec;
101:         }
102:         toflt(&val1);
103:     }
104:     else toflt(&val2);
105:     forever {
106:         if (val1.value.flt >= val2.value.flt)
107:             return (CELLP)nil;
108:         if (args == (CELLP)nil)
109:             return (CELLP)t;
110:         val1.value.fix = val2.value.fix;
111:         args = getlarg(args, &val2);    ec;
112:         toflt(&val2);
113:     }
114: }
115:
116: CELLP member_f(args)
117: CELLP args;
118: {
119:     CELLP   factor, error();
120:
121:     if (args->id != _CELL || args->cdr->id != _CELL)
122:         return error(NEA);
123:     factor = args->car;
124:     args = args->cdr->car;
125:     while(args->id == _CELL) {
126:         if (equal(factor, args->car)) ………equalを使ってリストの要素を比較していく
127:             return args;
128:         args = args->cdr;
129:     }
130:     return (CELLP)nil;
131: }
```

List 6.35 str.c 新しいファイルを作成

```
 1: /*                                  */
 2: /*           STR                    */
 3: /*                                  */
 4: /*       string oprations           */
 5: /*                                  */
 6:
 7: #include    "lisp.h"
 8: #include    <ctype.h>
 9: #define     forever     for(;;)
10:
11: CELLP ascii_f(arg)
12: CELLP arg;
13: {
14:     char    tp[2];
15:     CELLP   cp, error();
16:     NUMP    newnum();
17:     ATOMP   old_atom(), mk_atom();
18:
19:     if (arg->id != _CELL)   return error(NEA);
20:     if (arg->car->id == _ATOM) {                       ┐
21:         *tp = *(((ATOMP)(arg->car))->name);            │
22:         cp = (CELLP)newnum();    ec;                   │ シンボル → 数値アトム
23:         ((NUMP)cp)->value.fix = (long)*tp;             │
24:         return cp;                                     │
25:     }                                                  ┘
26:     else if (arg->car->id == _FIX) {                               ┐
27:         *tp = (char)(((NUMP)(arg->car))->value.fix);               │
28:         tp[1] = '¥0';                                              │
29:         if ((cp = (CELLP)old_atom(tp)) == NULL)                    │ 数値アトム → シンボル
30:         cp = (CELLP)mk_atom(tp);                                   │
31:         return cp;                                                 │
32:     }                                                              ┘
33:     return error(IAAN);
34: }
35:
36: CELLP implode_f(args)
37: CELLP args;
38: {
39:     char    c, nambuf[NAMLEN];
40:     ATOMP   ap, old_atom(), mk_atom();
41:     CELLP   error();
42:
43:     if (args->id != _CELL)
44:         return error(NEA);
45:     if ((args = args->car)->id != _CELL)
46:         return error(IAL);
47:     *nambuf = '¥0';                                                      ┐
48:     while (args != (CELLP)nil) {                                         │
49:         if ((c = args->car->id) != _ATOM)                                │ 引数のシンボルの印字名を順
50:             return error(IAA);                                           │ に nambuf につなぎ足していく
51:         if (strapnd(nambuf, ((ATOMP)(args->car))->name))                 │
52:             break;                                                       │
53:         args = args->cdr;                                                ┘
54:     }                                                  ┐
55:     if ((ap = old_atom(nambuf)) == NULL)               │ シンボルを作るときの
56:         return (CELLP)mk_atom(nambuf);                 │ パターンとなる手順
57:     return (CELLP)ap;                                  ┘
58: }
59:
60: CELLP explode_f(arg)
61: CELLP arg;
62: {
63:     char    nam[3], str[50];
64:     STR     s = str;
65:     STR     _nam;
66:     ATOMP   ap, old_atom(), mk_atom();
```

```
 67:     CELLP   cp1, cp2, error(), newcell();
 68:
 69:     if (arg->id != _CELL)   return error(NEA);
 70:     switch ((arg = arg->car)->id) {
 71:         case _CELL: return error(IAAN);
 72:         case _ATOM: s = ((ATOMP)arg)->name;
 73:                 break;
 74:         case _FIX:  sprintf(str, "%ld", ((NUMP)arg)->value.fix); ┐ 数値はsprintfで
 75:                 break;                                           │ 文字になおす
 76:         case _FLT:  sprintf(str, "%g", ((NUMP)arg)->value.flt); ┘
 77:     }
 78:     stackcheck;
 79:     *(++sp) = cp1 = newcell();  ec;
 80:     forever {
 81:         _nam = nam;
 82:         if (iskanji(*s))     ┐
 83:             *_nam++ = *s++;  │ 1文字nam[]にCopyする
 84:         *_nam++ = *s++;      │
 85:         *_nam = '¥0';        ┘
 86:         if ((ap = old_atom(nam)) == NULL)
 87:             ap = mk_atom(nam);       ┐
 88:         if (*s != '¥0') {            │
 89:             cp2 = newcell();    ec;  │
 90:             cp1->car = (CELLP)ap;    │ 作ったシンボルをリストにつなぐ
 91:             cp1->cdr = cp2;          │
 92:             cp1 = cp2;               │
 93:             continue;                ┘
 94:         }
 95:         cp1->car = (CELLP)ap;
 96:         return *(sp--);
 97:     }
 98: }
 99:
100: CELLP alen_f(arg) ………文字列の長さを返す
101: CELLP arg;
102: {
103:     CELLP   error();
104:     NUMP    np, newnum();
105:
106:     if(arg->id != _CELL)    return error(NEA);
107:     if((arg = arg->car)->id != _ATOM)   return error(IAA);
108:     np = newnum();  ec;
109:     np->value.fix = (long)strlen( ((ATOMP)arg)->name);
110:     return (CELLP)np;
111: }
112:
113: static strapnd(str1, str2) ………str2をstr1につぎ足す.
114: STR str1, str2;               ただし，全体の長さはNAMLENを超えないようにする
115: {
116:     STR base = str1;
117:
118:     while (*str1 != '¥0')   ++str1;
119:     while (*str2 != '¥0') {
120:         *str1++ = *str2++;
121:         if (str1 - base == NAMLEN - 1) {
122:             *str1 = '¥0';
123:             return TRUE;
124:         }
125:     }
126:     *str1 = '¥0';
127:     return FALSE;
128: }
```

6.5 機能拡張(5) 汎関数と関数引数の追加 —MADALTOバージョン—

ここではマップ関数に代表されるいくつかの汎関数を追加します．これに伴い，fexpr 型の関数を定義できるようにします．また trace の機能も加わります．

6.5.1 *eval* と *apply*

eval と *apply* は，Lisp インタープリタそのものの機能を関数として取り出したものです．*eval* は，評価をおこない，*apply* は，関数の適用をおこないます．

```
% (eval '(car '(a b)))
a

% (setq a 'b)
b

% (eval 'a)
b
```

このように，*eval* はインタープリタ自体とまったく同様に評価をおこないます．また，引数として，環境リストを渡してやると，その環境の下で評価をおこないます．

```
% (eval '(cons a b) '((b the fantasia) (a . van)))
(van the fantasia)
```

この例では a に van が，b に (the fantasia) が束縛された環境の下で (cons a b) を評価しています．

```
% (apply 'cons '(a b))
(a . b)
```

apply にはこのように関数と，引数のリストを与えます．この時，引数リスト自体は評価されますが，その中の引数ひとつひとつが再び評価されることはありません．*apply* にも，引数として関数実行時に使用する環境リストを与えることができます．

```
% (apply '(lambda (x) (cons y x)) '(tictaco) '((y . tama)))
(tama . tictaco)
```

この例では，yにtamaが束縛されている環境の下で,関数(lambda(x)(cons　y　x))がtictacoに適用されています．

*apply*の引数の与え方は幾分わかりにくいので，funcallという関数も用意されています．

```
% (funcall 'cons 'a 'b)
(a . b)
```

ただし，*funcall*には評価環境を与えることはできません．

このように，関数を，引数として，すなわちデータとして扱うようになると，関数が定義される時の環境と,関数が実行される時の環境に違いが生じる可能性が出てきます.(「2.3.4 function式によるクロージャ」参照)そこで，このバージョンでfunction式を導入することにします．

ここで，function式の効能をもう1度説明しておきます．

```
% (de zdo (z y) (funcall z y))
zdo

% (zdo 'car '(martin))
martin
```

*zdo*はfuncallと同じ働きをする関数です．

```
% (setq y 'Creamy)
Creamy

% (de tam (x) (cons y x))
tam

% (tam 'tam)
(Creamy . tam)
```

*tam*はyの値を引数に*cons*する関数です．いまyの値はCreamyになっているので，*tam*は3番目の例のように働きます．ところが，これを*zdo*に渡すとおかしくなります．

```
% (zdo 'tam 'tam)
(tam . tam)
```

これは,*zdo*の引数にyがあり，これに*zdo*の実引数tamがbindされているためです．*zdo*に関数*tam*を渡す時にfunction式を使えば期待どおりの動作をしてくれます．

```
% (zdo (function tam) 'tam)
(Creamy . tam)
```

function は，引数である関数に，その時点での環境を添えたものを返します．*funcall* がこれを受け取ると，付随している環境を取り出して，その下で関数を実行します．この次のMAXバージョンでは *function* 式は略記することができ，上の例は通常次のように略記します．

```
% (zdo #'tam 'tam)
(Creamy . tam)
```

関数を引数として扱う場合は，かならず，〝#'〟をつけて渡すことを習慣とするとよいでしょう．

6.5.2 fexpr関数

このバージョンから引数を評価しないタイプの関数, fexprをユーザーが定義できるようになります．これを定義するには，関数 *df* を使います．たとえば，これを使ってCのcase文のような働きをするものを作ってみましょう．

```
% (df case (key . bodies)
%       (prog ()
%             (setq key (eval key))
%        loop (cond ((equal 1 key) (return (eval (car bodies)))))
%             (setq key (sub1 key))
%             (setq bodies (cdr bodies))
%             (go loop)))
case

% (setq n 2)
2

% (case n (car '(a b)) (cons 'a 'b) (print 'zndoko))
(a . b)
```

case に渡された引数の中には, *print* を含むものがありますが, *case* が返した値以外のものは出力されていません．これは，*print* を含むS式が評価されていないからです．このように，制御構造を作ろうとする時には，引数が評価されないことが重要になります．

fexpr の実体は nlamdba 式です．nlambda 式とは，

(nlambda <引数リスト>　<本体>)

というように，lambda 式の第 1 要素が nlambda に変わっただけのものです．しかし，これだけのことで，引数は全然評価されなくなります．nlambda は lambda 式と同様に単独でも関数として使用できます．fexpr 関数を表すアトムにはアトム構造体のメンバ fptr に nlambda 式が入っています．

6.5.3　マップ関数

普通の Lisp に備わっているマップ関数には Table 6.3 に示す 6 種類があります．

		返すリストの作成方法		
		元のリスト	値のリスト	append
関数に渡される引数	要素	mapc	mapcar	mapcan
	部分リスト	map	maplist	mapcon

Table 6.3　マップ関数の分類

マップ関数は関数とリストを引数にとります．*mapcar* の場合，リストの各要素が関数に引数として与えられ，その関数が評価された結果がひとつのリストにまとめられ，*mapcar* の評価値になります．

```
% (de sqr (x) (times x x))
sqr

% (mapcar (function sqr) '(1 2 3 4))
(1 4 9 16)
```

sqr は，引数を 2 乗する関数です．これと数値を要素とするリストを *mapcar* に与えると，リストの各要素が 2 乗されたリストが返ってきています．

一方，*maplist* では，リストの要素ではなく，元のリスト，リストの cdr，リストの cdr の cdr，等が順に関数に渡され，その結果がリストにまとめられて返ってきます．

```
% (maplist (function reverse) '(a b c))
((c b a) (c b) (c))
```

この例では *reverse* には (a b c)，(b c)，(c) が渡されています．

mapcar と *maplist* の違いは，関数に渡される引数が，リストの要素であるか，部分リストであるか，という点です．これは引数を取り出すのに，与えられたリストの car をとった場合と cdr をとった場合と見ることができます．Table 6.3 で上段に書かれているマップ関数は要素を取り出すタイプのもので，下段にあるものは部分リストを関数に与えるタイプのものです．

次に，*mapcan* を見てみましょう．

```
% (de ifplus (x) (cond ((greaterp x 0) (list x)) (t nil)))
ifplus

% (ifplus 3)
(3)
```

ifplus は引数が正の数値の時には，その数値を要素とするリストを返し，それ以外の場合は nil を返します．これに，正負の数を含むリストを *mapcan* を使って与えると，次のようになります．

```
% (mapcan (function ifplus) '(5 -1 3 -2))
(5 3)
```

mapcan は，要素を関数に与えるタイプなので，それぞれの要素について *ifplus* が返す値は，5, −1, 3, −2 に対して，

```
(5)  nil  (3)  nil
```

となります．*mapcan* はこれらの値を *append* したものを返します．実際にこれらのリストを *append* に与えてみると，

```
% (append '(5) 'nil '(3) 'nil)
(5 3)
```

となります．もっとも，セルの使用効率のために，*mapcan* では *append* のようにリストのコピーはせず，関数が返した値をそのまま接続したものを返しています．

マップ関数は，返す値を作る方法によって3種類に分けられ，それぞれ，関数の返した値を要素とするリスト(Table 6.3 中)，関数の返した値を *append* したもの(Table 6.3 右)，引数として渡されたリストそのもの(Table 6.3 左)，を返すようになっています．最後のものは，主に，関数が返す値よりも関数の副作用が問題となる場合に使います．

6.5.4 トレース機能

トレース機能を実現するために，*trace* と *untrace* という関数を追加します．*trace* はトレースの開始を指示し，*untrace* は終了を指示します．

```
% (de fact (n) (cond ((zerop n) 1) (t (times n (fact (sub1 n))))))
fact

% (trace fact)
(fact)
```

trace にはトレースするべき関数を与えます．引数の数はいくつでも構いません．トレースを指示した関数のリストが値として返されます．

```
% (fact 3)
->fact : args 3
-->fact : args 2
--->fact : args 1
---->fact : args 0
<----fact : value 1
<---fact : value 1
<--fact : value 2
<-fact : value 6
6
```

トレースされている関数が呼ばれると，関数の名前と引数が表示され，また呼び出しの深さが矢印の長さで示されます．また関数の返した値も同様に表示されます．

```
% (untrace fact)
(fact)

% (fact 3)
6
```

untrace はトレースを終了させるほかは trace と同様に働きます．

6.5.5 汎関数処理の追加に伴うプログラムの変更

MONTINOバージョンに，マップ関数の処理を記述したmapper.cファイルが追加され，またプログラムリストに次のような変更が必要となります．

● lisp.h (List 6.36)

fexpr型関数を表すコードおよび関数のトレース指定を検出するためのマスクの2種類の定数が追加されます．また，マップ関数に関するエラーコードが加えられます．

● defvar.h/var.h (List 6.37/List 6.38)

function, funarg, nlambdaの3つのシステムアトムと，トレースの深さを記録するための大域変数が追加されます．

● error.c (List 6.40)

追加されたエラーに対応するエラーメッセージがerr_msg [] に加えられます．

● main.c (List 6.41)

reset_err()の中に，トレース関係の大域変数の初期化ルーチンが追加さます．またmk_sys_atoms()では，あらたに加わった3つのシステムアトムの初期化がおこなわれます．

● inisubr.c (List 6.42)

汎関数関連のLisp関数を定義するため，init5()が追加され，ini_subr()も変更されます．

6.5.6 eval.cの変更 (List 6.39)

● eval_f() (64行〜84行)

eval()を使って，引数を評価します．eval()を呼ぶために，環境が必要となるので，fsubrとしてありますが，引数は評価されなければならないので，引数リストから取り出した引数を先にeval()で評価します．ソフトウェアスタックを2つ使い，＊(sp−1)に与えられた環境を，＊spに評価するべきS式を置いています．

● eval_arg_p() (1行〜14行)

lambda 式が, function 式の中に入っている場合にも, 引数が評価されるようにするため, function 式の中を調べるようにします.

● apply_f() (86行〜109行)

主要な処理は apply()を呼んで済ませています. apply()を呼ぶために, 環境が必要となるので, fsubr にしてあります. しかし, 引数は評価されなければならないので, 引数リストから取り出した引数は, eval()で評価してから使用しています. ソフトウェアスタックを3つ使い, *sp に関数, *(sp−1)に引数リスト, *(sp−2)に環境を置いています.

● apply() (16行〜62行)

関数が fexpr 型の場合と, funarg 式の場合の処理を追加します. 引数の評価は eval()の管轄なので, apply()では, fexpr は expr と同様に扱います. したがって, 関数がリストで与えられ, その car が nlambda である場合は, lambda である場合と同じ処理をおこないます. 一方, 関数の car が funarg である場合は, funarg 式と判断して, この中から関数と環境を取り出してこれまでのものに置き換えます.

また, トレースをおこなうために数行プログラムを追加します. apply()を呼び出す前後で関数の fntype を調べ, trace ビット(_TR)が立っていれば, トレース処理をおこないます. 具体的なトレース処理は関数 trarg()と trval()がおこなっています.

● trval() (111行〜124行)

関数が返した値をトレース表示します. トレースの深さを記録するためにグローバル変数 trind を使います. ここでは関数が値を返しているので, 関数トレースの深さをひとつ減らします. 矢印の長さは trind の値で決めています.

● trargs() (126行〜149行)

これから実行する関数と引数を表示します. 矢印の長さは trind の値で決めています. また, 引数は引数リストの要素を順に表示しています. 引数の最後のセルの cdr が nil でない時には, それも表示するようになっています.

● trace_f() (151行〜170行)

関数を表すアトムから fntype を取り出し, _TR と or をとって trace ビットを立てます. eval()は trace ビットが立っているかどうかを調べ, on になっているものについてはトレース処理をおこないます.

● untrace_f()　(172 行〜188 行)

関数を表すアトムから fntype を取り出し，_TR を使って trace ビットを降ろします．

● funcall_f()　(190 行〜200 行)

実際の処理は apply() を呼んで済ませています．apply() を呼ぶために，環境リストが必要なので，fsubr にしてあります．そのため，関数，引数リストを eval() を使って評価してから apply() に渡すようにしています．

6.5.7　control.c の変更　(List 6.43)

● df_f()　(1 行〜15 行)

これは de_f() とほとんど同じで，lambda 式の代わりに nlambda 式を作るところが異なるだけです．

● funct_f()　(17 行〜29 行)

function の定義を与える関数です．fsubr なので，現在の環境を受け取ることができ，これをアトム funarg，関数本体とともにひとつのリストにまとめて返しています．

6.5.8　mapper.c の追加　(List 6.44)

Will o'Lisp では，Table 6.1 に示した 6 種類のマップ関数のうち，*mapcar* と *mapcan* そして *mapcon* を実装しました．この 3 つの作り方がわかっていれば，残りの 3 種類を作るのも容易でしょう．

● mapargchk()　(8 行〜19 行)

マップ関数の引数の取り出しとタイプチェックをおこないます．これを呼ぶ側の関数では，取り出した引数をおくための変数のアドレスを与えるようにします．マップ関数はすべて同じ形式の引数をとるので，後から 3 つの関数を追加する時にもこれを使うことができます．

● mapcar_f()　(21 行〜48 行)

これは，リストの各要素を関数に与え，結果をリストにまとめて返すタイプです．リストを作る部分は今までと同様で，＊sp に先頭を置いています．引数に関数を適用する部分は，どのマップ関数も同じで，apply() を使います．しかし，apply() は単独の引数ではなく，引数リストを要求しますから，＊(sp−1) にセルを持ってきて，このセルに順番に各要素を入れています．

● mapcon_f()（50行〜92行）

mapcon は引数リストの cdr に対し順番に関数を適用し，*nconc* でつなぐため，関数によっては副作用が残ります．

```
% (setq x '((a b) (c d) (e f)))
((a b) (c d) (e f))

% (mapcon 'car x)
(a b c d e f)

% x
((a b c d e f) (c d e f) (e f))
```

この例はまだよいほうで，場合によっては循環リストが作成されてしまいます．

```
% (mapcon 'print '(a b c))
(a b c)
(b c)
(c)
(a b c c c ……
```

これは，(b c)の末尾を自分自身の一部である(c)につなぎ換えてしまったためです．この例の結果は *mapcon* の仕様の上では正しいわけですが，これを C で実現するには工夫が必要です．安直に結果を *nconc* でつないでしまうと，引数のリストが循環リストになってしまい，いつまでも *mapcon* の実行が終わらなくなってしまうからです．そこで，mapcon_f()では，それぞれの cdr 部に関数を適用した結果をリストの最後尾のセルを次々とスタック上に蓄えていき，すべての結果が得られた後で，それらのセルの cdr を書き換えています．

● mapcan_f()（94行〜137行）

これは，リストの各要素を関数に与え，結果を *append* して返すタイプです．*append* と同様，関数が返す値が nil でないものを最初にひとつ見つけ，これを先頭セルとして*sp に置きます．後はこれにつないでいくだけです．

List 6.36 lisp.h 追加

```
1: #define _FEXPR  0       /* 0000 */
2: #define _TR     0x10
3: #define MOL     42   /* Mapped Object must be a List */
```

List 6.37 defvar.h 追加

```
1: ATOMP   function, funarg, nlambda;
2: int     trind = 1;
```

List 6.38 var.h 追加

```
1: extern ATOMP    function, funarg, nlambda;
2: extern int      trind;
```

List 6.39 eval.c ファイルごと差し替え

```
 1: /*                                    */
 2: /*           EVAL and APPLY           */
 3: /*                                    */
 4:
 5: #include     "lisp.h"
 6:
 7: CELLP eval(form, env)
 8: CELLP form, env;
 9: {
10:     CELLP   cp, apply(), atomvalue(), evallist(), error();
11:     ATOMP   func;
12:
13:     switch (form->id) {
14:         case _ATOM:
15:             cp = atomvalue((ATOMP)form, env);
16:             break;
17:         case _FIX:
18:         case _FLT:
19:             return form;
20:         case _CELL:
21:             stackcheck;
22:             ++sp;
23:             func = (ATOMP)form->car;
24:             if (eval_arg_p(func)) {
25:                 *sp = evallist(form->cdr, env);
26:                 if (err) break;
27:             }
28:             else
29:                 *sp = form->cdr;
30:             cp = apply((CELLP)func, *sp, env);
31:             sp--;
32:             break;
33:         default:
34:             error(ULO);
35:     }
36:     if (err == ERR) {
37:         pri_err(form);
38:         return NULL;
39:     }
40:     return cp;
41: }
42:
43: static int eval_arg_p(func)
44: CELLP func;
```

```
 45: {
 46:     if (func->id == _ATOM && ((ATOMP)func)->ftype & _EA)
 47:         return TRUE;
 48:     if (func->id == _CELL && func->car == (CELLP)lambda)
 49:         return TRUE;
 50:     if (func->id == _CELL ………funarg式の場合はその中の関数について再帰呼び出しする
 51:         && func->car == (CELLP)funarg
 52:         && func->cdr->id == _CELL)
 53:         return eval_arg_p(func->cdr->car);
 54:     return FALSE;
 55: }
 56:
 57: CELLP apply(func, args, env)
 58: CELLP func, args, env;
 59: {
 60:     CELLP   (*funcp)(), bodies, result = (CELLP)nil;
 61:     CELLP   bind(), error();
 62:     char    funtype = ~_TR;………トレースビットを消しておく
 63:
 64:     switch (func->id) {
 65:         case _ATOM:
 66:             funtype = ((ATOMP)func)->ftype;
 67:             if (funtype & _UD)
 68:                 return error(UDF);
 69:             if (funtype & _TR)………トレースビットが立っていたら引数を表示する
 70:                 trarg((ATOMP)(*++sp = func), args);
 71:             if (funtype & _SR) {
 72:                 funcp = (CELLP (*)())((ATOMP)func)->fptr;
 73:                 if (funtype & _EA)
 74:                     result = (*funcp)(args);
 75:                 else
 76:                     result = (*funcp)(args, env);
 77:                 break;
 78:             }
 79:             func = ((ATOMP)func)->fptr;
 80:         case _CELL:
 81:             if (func->cdr->id != _CELL)
 82:                 return error(IFF);              funarg式の場合は,関数と環境をfunarg式
 83:             if (func->car == (CELLP)funarg)………の中のものでおきかえて再帰呼び出しする
 84:                 result = apply(func->cdr->car. args. func->cdr->cdr->car);
 85:             else if (func->car == (CELLP)lambda
 86:                     || func->car == (CELLP)nlambda) {
 87:                 bodies = func->cdr->cdr;
 88:                 stackcheck;
 89:                 *++sp = bind(func->cdr->car. args, env); ec;
 90:                 for (; bodies->id == _CELL; bodies = bodies->cdr) {
 91:                     result = eval(bodies->car, *sp); ec;
 92:                 }
 93:                 --sp;
 94:             }
 95:             break;
 96:         default:
 97:             return error(IFF);
 98:     }
 99:     ec;
100:     if (funtype & _TR)………トレースビットが立っていたら値を表示する
101:         trval((ATOMP)*sp--, result);
102:     return result;
103: }
104:
105: static CELLP evallist(args. env)
106: CELLP args, env;
107: {
108:     CELLP   cpl. newcell(). eval();
109:
110:     if (args->id != _CELL)
111:         return (CELLP)nil;
```

```
112:     stackcheck;
113:     *++sp = newcell(); ec;
114:     cp1 = *sp;
115:     cp1->car = eval(args->car, env); ec;
116:     args = args->cdr;
117:     while (args->id == _CELL) {
118:         cp1->cdr = newcell(); ec;
119:         cp1 = cp1->cdr;
120:         cp1->car = eval(args->car, env); ec;
121:         args = args->cdr;
122:     }
123:     cp1->cdr = (CELLP)nil;
124:     return *sp--;
125: }
126:
127: CELLP bind(keys, values, env)
128: CELLP keys, values, env;
129: {
130:     CELLP   push(), error();
131:
132:     if (keys != (CELLP)nil && keys->id == _ATOM) {
133:         env = push(keys, values, env); ec;
134:         return env;
135:     }
136:     stackcheck;
137:     *++sp = env;
138:     while (keys->id == _CELL) {
139:         if (values->id != _CELL)
140:             return error(NEA);
141:         *sp = push(keys->car, values->car, *sp); ec;
142:         keys = keys->cdr;
143:         values = values->cdr;
144:     }
145:     if (keys != (CELLP)nil && keys->id == _ATOM) {
146:         *sp = push(keys, values, *sp); ec;
147:     }
148:     return *sp--;
149: }
150:
151: static CELLP push(key, value, env)
152: CELLP key, value, env;
153: {
154:     CELLP   newcell();
155:
156:     stackcheck;
157:     *++sp = newcell(); ec;
158:     (*sp)->cdr = env;
159:     env = *sp;
160:     env->car = newcell(); ec;
161:     env->car->car = key;
162:     env->car->cdr = value;
163:     return *sp--;
164: }
165:
166: static CELLP atomvalue(ap, env)
167: ATOMP ap;
168: CELLP env;
169: {
170:     CELLP   error();
171:
172:     while (env->id == _CELL) {
173:         if (env->car->id != _CELL)
174:             return error(EHA);
175:         if (env->car->car == (CELLP)ap)
176:             return env->car->cdr;
177:         env = env->cdr;
178:     }
```

```
179:     return ap->value;
180: }
181:
182: CELLP eval_f(args, env)
183: CELLP args, env;
184: {
185:     CELLP   result, error(), eval();
186:
187:     if (args->id != _CELL)
188:         return error(NEA);
189:     sp += 2;
190:     stackcheck;
191:     if (args->cdr->id == _CELL) {………引数として環境リストが与えられた場合
192:         *(sp-1) = eval(args->cdr->car, env); ec;
193:         *sp = eval(args->car, env); ec;
194:         result = eval(*sp, *(sp-1));
195:     }
196:     else {………環境リストが与えられなかった場合
197:         *sp = eval(args->car, env); ec;
198:         result = eval(*sp, env);
199:     }
200:     sp -= 2;
201:     return result;
202: }
203:
204: CELLP apply_f(args, env)
205: CELLP args, env;
206: {
207:     CELLP result, error(), eval(), apply();
208:
209:     if (args->id != _CELL || args->cdr->id != _CELL)
210:         return error(NEA);
211:     sp += 3;
212:     stackcheck;
213:     *sp = eval(args->car, env); ec;
214:     *(sp-1) = eval(args->cdr->car, env); ec;
215:     if ((*(sp-1))->id != _CELL && *(sp-1) != (CELLP)nil)
216:         return error(IAL);
217:     if (args->cdr->cdr->id == _CELL) {
218:         *(sp-2) = eval(args->cdr->cdr->car, env); ec;………引数として環境リストが
219:     }                                                     与えられた場合
220:     else
221:         *(sp-2) = env;………環境リストが与えられなかった場合
222:     result = apply(*sp, *(sp-1), *(sp-2));
223:     sp -= 3;
224:     return result;
225: }
226:
227: static trval(func, val)………トレース時に関数が返した値を表示する
228: ATOMP func;
229: CELLP val;
230: {
231:     int i;
232:
233:     --trind;
234:     fputc('<', cur_fpo);
235:     for (i = 0; i<trind; i++)
236:         fputc('-', cur_fpo);
237:     fprintf(cur_fpo, "%s : value ", func->name);
238:     print_s(val, ESCON);
239:     fputc('¥n', cur_fpo);
240: }
241:
242: static trarg(func, args)………トレース時に関数に与えられた引数を表示する
243: ATOMP func;
244: CELLP args;
245: {
```

```
246:     int i;
247:
248:     for (i = 0; i<trind; i++)
249:         fputc('-', cur_fpo);
250:     fprintf(cur_fpo, ">%s : args ", func->name);
251:     if (args->id != _CELL)
252:         return;
253:     while (args->cdr->id == _CELL) {
254:         print_s(args->car, ESCON); ec;
255:         fputc(' ', cur_fpo);
256:         args = args->cdr;
257:     }
258:     print_s(args->car, ESCON); ec;
259:     if (args->cdr != (CELLP)nil) {
260:         fprintf(cur_fpo, " . ");
261:         print_s(args->cdr, ESCON); ec;
262:     }
263:     fputc('¥n', cur_fpo);
264:     ++trind;
265: }
266:
267: CELLP trace_f(args)
268: CELLP args;
269: {
270:     char    funtype;
271:     CELLP   error();
272:     CELLP   cp = args;
273:
274:     if (args->id != _CELL)
275:         return error(NEA);
276:     while (args->id == _CELL) {
277:         if (args->car->id != _ATOM)
278:             return error(IAA);
279:         funtype = ((ATOMP)args->car)->ftype;
280:         if (funtype & _UD)
281:             return error(UDF);
282:         ((ATOMP)args->car)->ftype = funtype | _TR;………トレースビットを立てる
283:         args = args->cdr;
284:     }
285:     return cp;
286: }
287:
288: CELLP untrace_f(args)
289: CELLP args;
290: {
291:     char    funtype;
292:     CELLP   cp = args;
293:
294:     if (args->id != _CELL)
295:         return error(NEA);
296:     while (args->id == _CELL) {
297:         if (args->car->id != _ATOM)
298:             return error(IAA);
299:         funtype = ((ATOMP)args->car)->ftype;
300:         ((ATOMP)args->car)->ftype = funtype & ~_TR;………トレースビットを降ろす
301:         args = args->cdr;
302:     }
303:     return cp;
304: }
305:
306: CELLP funcall_f(args, env)
307: CELLP args, env;
308: {
309:     ATOMP func;
310:
311:     if (args->id != _CELL)
312:         return error(NEA);
```

```
313:     func = (ATOMP)eval(args->car, env); ec;
314:     args = evallist(args->cdr, env); ec;
315:     return apply((CELLP)func, args, env);
316: }
```

List 6.40 error.c err_msg[]へメッセージを追加

```
  1:     "Mapped Object must be a List",
```

List 6.41 main.c reset_err(), mk_sys_atoms(), greeting()を変更

```
  1: static reset_err()
  2: {
  3:     cur_fpi = stdin;
  4:     cur_fpo = stdout;
  5:     err = NONERR;
  6:     txtp = oneline;
  7:     *txtp = '¥0';
  8:     for (sp = stacktop; sp < stacktop + STACKSIZ; ++sp)
  9:         *sp = (CELLP)nil;
 10:     sp = stacktop - 1;
 11:     verbos = ON;
 12:     throwlabel = throwval = (CELLP)nil;
 13:     prompt = stdprompt;
 14:     trind = 1; ………トレース時のインデントの深さを1にもどす
 15: }
 16:
 17: static mk_sys_atoms()
 18: {
 19:     ATOMP   mk_atom();
 20:     CELLP   quote_f(), funct_f();
 21:
 22:     mk_nil();
 23:     t = mk_atom("t");
 24:     lambda = mk_atom("lambda");
 25:     eofread = mk_atom("EOF");
 26:     prompt = stdprompt = mk_atom("% ");
 27:     quote = mk_atom("quote");
 28:     quote->ftype = _FSUBR;
 29:     quote->fptr = (CELLP)quote_f;
 30:     nlambda = mk_atom("nlambda");
 31:     function = mk_atom("function");
 32:     function->ftype = _FSUBR;
 33:     function->fptr = (CELLP)funct_f;
 34:     funarg = mk_atom("funarg");
 35: }
 36:
 37: static greeting()
 38: {
 39:     fprintf(stdout,"¥n");
 40:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
 41:     fprintf(stdout,"¥t        M A D A L T O¥n");
 42:     fprintf(stdout,"¥t  Will o'Lisp Version 0.90¥n");
 43:     fprintf(stdout,"¥t       (C)  1986, Mar¥n¥n");
 44:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
 45: }
```

List 6.42 inisubr.c ini_subr()を変更，init5()を追加

```
 1: ini_subr()
 2: {
 3:     init0();
 4:     init1();
 5:     init2();
 6:     init3();
 7:     init4();
 8:     init5();
 9: }
10:
11: static init5()
12: {
13:     CELLP eval_f(), apply_f(), funcall_f();
14:     CELLP df_f(), trace_f(), untrace_f();
15:     CELLP mapcar_f(), mapcon_f(), mapcan_f();
16:
17:     defsubr("eval",     eval_f,     _FSUBR);
18:     defsubr("apply",    apply_f,    _FSUBR);
19:     defsubr("funcall",  funcall_f,  _FSUBR);
20:     defsubr("df",       df_f,       _FSUBR);
21:     defsubr("trace",    trace_f,    _FSUBR);
22:     defsubr("untrace",  untrace_f,  _FSUBR);
23:     defsubr("mapcar",   mapcar_f,   _SUBR);
24:     defsubr("mapcon",   mapcon_f,   _SUBR);
25:     defsubr("mapcan",   mapcan_f,   _SUBR);
26: }
```

List 6.43 control.c df_f(), funct_f()を追加

```
 1: CELLP df_f(args, env)
 2: CELLP args, env;
 3: {
 4:     CELLP val, cons(), error();
 5:     ATOMP func;
 6:
 7:     if (args->id != _CELL || args->cdr->id != _CELL)
 8:         return error(NEA);
 9:     if ((func = (ATOMP)args->car)->id != _ATOM)
10:         return error(IAA);
11:     val = cons((CELLP)nlambda, args->cdr); ec;
12:     func->ftype = _FEXPR;
13:     func->fptr = val;
14:     return (CELLP)func;
15: }
16:
17: CELLP funct_f(args, env) ………関数functionの実体
18: CELLP args, env;
19: {
20:     CELLP cons(), error();
21:
22:     if (args->id != _CELL)
23:         return error(NEA);
24:     stackcheck;
25:     *++sp = cons(env, (CELLP)nil); ec;
26:     *sp = cons(args->car, *sp); ec;
27:     *sp = cons((CELLP)funarg, *sp); ec;
28:     return *sp--;
29: }
```

List 6.44 mapper.c 新しいファイルの作成

```
 1: /*                   */
 2: /*      MAPPER       */
 3: /*                   */
 4: /*                   */
 5:
 6: #include    "lisp.h"
 7:
 8: static CELLP mapargchk(args, funcp, listp) ……マップ関数の引数の数とタイプが適切かどうか
 9: CELLP args, *funcp, *listp;                   調べる．また，与えられた関数とリストをfuncp,
10: {                                             listpで指定された変数にとりだす
11:     CELLP error();
12:
13:     if (args->id != _CELL || args->cdr->id != _CELL)
14:         return error(NEA);
15:     if ((*funcp = args->car)->id != _ATOM && (*funcp)->id != _CELL)
16:         return error(IAAL);
17:     if ((*listp = args->cdr->car)->id != _CELL && *listp != (CELLP)nil)
18:         return error(IAL);
19: }
20:
21: CELLP mapcar_f(args)
22: CELLP args;
23: {
24:     CELLP list, func, cp;
25:     CELLP cons(), error(), newcell(), apply();
26:
27:     mapargchk(args, &func, &list); ec;
28:     if (list == (CELLP)nil)
29:         return (CELLP)nil;
30:     sp += 2;
31:     stackcheck;
32:     *sp = newcell(); ec;
33:     cp = *sp;
34:     *(sp-1) = cons(list->car, (CELLP)nil); ec;
35:     cp->car = apply(func, *(sp-1), (CELLP)nil); ec;
36:     list = list->cdr;
37:     while (list->id == _CELL) {
38:         (*(sp-1))->car = list->car;
39:         cp->cdr = newcell(); ec;
40:         cp = cp->cdr;
41:         cp->car = apply(func, *(sp-1), (CELLP)nil); ec;
42:         list = list->cdr;
43:     }
44:     cp->cdr = (CELLP)nil;
45:     cp = *sp;
46:     sp -= 2;
47:     return cp;
48: }
49:
50: CELLP mapcon_f(args)
51: CELLP args;
52: {
53:     CELLP arg, *sp1, *sp2, *sp3, *re_sp, list, func;
54:     CELLP cons(), newcell(), apply(), error();
55:
56:     mapargchk(args, &func, &list); ec;
57:     if (list == (CELLP)nil)
58:         return (CELLP)nil;
59:     stackcheck;
60:     *++sp = cons(list, (CELLP)nil); ec; ………与えられたリスト全体を要素とする引数リストを
61:     arg = *sp;                                つくり，変数argに置く
62:     stackcheck;
63:     *++sp = apply(func, arg, (CELLP)nil); ec; ………関数を適用した結果をスタックに積む
64:     re_sp = sp;
65:     if ((*sp)->id != _CELL) ………結果がリストでなければ捨てる
66:         sp--;
67:     list = list->cdr;
```

```
 68:      while (list->id == _CELL) {
 69:          arg->car = list;
 70:          stackcheck;
 71:          *++sp = apply(func, arg, (CELLP)nil); ec;
 72:          if ((*sp)->id != _CELL)
 73:              sp--;
 74:          list = list->cdr;
 75:      }
 76:      if (sp<re_sp) { ………ひとつも結果がたまっていなければnilを返す
 77:          sp = re_sp-2;
 78:          return (CELLP)nil;
 79:      }
 80:      sp1 = re_sp;
 81:      sp2 = sp;
 82:      while (sp1<sp2) { ………スタックに蓄積された結果(すべてリスト)の，最後尾のセルをスタックにつむ
 83:          stackcheck;
 84:          for (*++sp = *sp1++; (*sp)->cdr->id == _CELL; *sp = (*sp)->cdr);
 85:      }
 86:      sp1 = re_sp+1;
 87:      sp3 = sp2+1;
 88:      while (sp1 <= sp2) ………結果のリストを接続する
 89:          (*sp3++)->cdr = *sp1++;
 90:      sp = re_sp-2;
 91:      return *re_sp;
 92: }
 93:
 94: CELLP mapcan_f(args)
 95: CELLP args;
 96: {
 97:      CELLP cp, list, func;
 98:      CELLP error(), cons(), apply();
 99:
100:      mapargchk(args, &func, &list); ec;
101:      if (list == (CELLP)nil)
102:          return (CELLP)nil;
103:      sp += 2;
104:      stackcheck;
105:      *(sp-1) = cons(list->car, (CELLP)nil); ec;
106:      *sp = apply(func, *(sp-1), (CELLP)nil); ec;
107:      while ((*sp)->id != _CELL) { ………mapcanが返すリストの先頭となるべきセルを見つける
108:          if (*sp != (CELLP)nil)
109:              return error(MOL);
110:          if (list->cdr->id != _CELL) {
111:              sp -= 2;
112:              return (CELLP)nil;
113:          }
114:          list = list->cdr;
115:          (*(sp-1))->car = list->car;
116:          *sp = apply(func, *(sp-1), (CELLP)nil); ec;
117:      }
118:      cp = *sp;
119:      while (cp->cdr->id == _CELL) ………関数の適用結果のリストの最後尾のセルを見つける
120:          cp = cp->cdr;
121:      if (cp->cdr != (CELLP)nil)
122:          return error(MOL);
123:      list = list->cdr;
124:      while (list->id == _CELL) {
125:          (*(sp-1))->car = list->car;
126:          cp->cdr = apply(func, *(sp-1), (CELLP)nil); ………
127:          while (cp->cdr->id == _CELL)
128:              cp = cp->cdr;
129:          if (cp->cdr != (CELLP)nil)
130:              return error(MOL);
131:          list = list->cdr;
132:      }
133:      cp->cdr = (CELLP)nil;
134:      cp = *sp;
135:      sp -= 2;
136:      return cp;
137: }
```

（68～75行の注釈）リストの各cdrに関数を適用した結果をスタックに蓄積する．ただし，結果がリストでない時は捨てる

（126行の注釈）与えられたリストの各carに関数を適用し，その結果得られたリストを，その前のリストの最後尾のセルのcdrにつなぐ

6.6 機能拡張(6) マクロ形式の追加
—MAXバージョン—

このバージョンではマクロに関する機能拡張が主題になります．まず，read.c ファイルにおける変更部分を見ることにします．

6.6.1 oblist に登録されないシンボルアトム

dm によって定義されたマクロを評価する時には，展開された式でシンボルアトムの衝突が起こらないように注意しなくてはなりません．たとえば，次のような式を評価することを考えてみます．

```
% (dm switch (index plusform zeroform minusform)
%     ^(let ((var ,index))
%           (cond ((greaterp var 0) ,plusform)
%                 ((zerop var) ,zeroform)
%                 (t ,minusform)))
switch
```

このマクロは，第１引数の符号によって場合分け処理をおこなう，FORTRAN の算術 if 文にあたるものです．

この定義では第１引数を何回も評価するのがいやだという理由で，第１引数の評価の結果を let を用いて変数 var に束縛し，１回だけの評価ですませようとしたようですが，さて，どうでしょうか．これでは思わぬ動作をする可能性があります．たとえば，

var＞5 …… var を返す
var≦5 …… nil を返す

という処理をおこなうつもりで，次のような式を評価します．

```
% (let ((var 6)) (switch (plus var −5) var nil nil))
1
```

最初に５よりも大きな６という値を var に設定していますから，この評価値は var の値がそのまま返されて６となるはずでした．ところが，返ってきたのはごらんのとおり，１という値です．というのも，この式が次のように展開されたからです．

```
(let ((var  (plus var  - 5))
     (cond  ((greaterp var 0) var)
            ((zerop var) nil)
            (t nil)))
```

マクロ展開の中で局所的に用いたかったはずのvarが，たまたまマクロの外で使われていた変数と衝突してしまったのです．すなわち，マクロの中で局所変数を使いたい時は決して他と重ならないような変数名でなくては安全に使えないということです．しかし，他と重ならないようなシンボルアトムとはいかなるものでしょうか．どんな奇想天外な印字名を使っても，重なる可能性がないわけではありません．そこで次のような関数が用意されています．

● gensym_f() (List 6.48 122行〜132行)

関数gensymは，最初に呼び出された時〝g000〟という印字名を持つシンボルアトムを作り出します．2回目には，〝g001〟，3回目には〝g002〟といった具合に，*gensym*の作り出すシンボルアトムは変わっていきます．これらのシンボルアトムはoblistに登録されていないシンボルアトムです．oblistに登録されていない新しいシンボルアトムは，それ以前に存在するどのS式からも参照を受けていないので，他との衝突は起こりません．

この*gensym*を用いて，先ほどの*switch*を書き直してみます．

```
% de switch (index plusform zeroform minusform)
%   (let ((var (gensym)))
%     ^(let ((,var ,index))
%          (cond ((greaterp ,var 0) ,plusform)
%                ((zerop ,var) ,zeroform)
%                (t ,minusform)))))
```

初めにvarを*gensysm*で作られるシンボルアトムに束縛し，マクロ展開はvarが展開式中に残らないようにしています．先ほどの例をこれでもう1度評価してみましょう．

```
% (let ((var 6)) (switch (plus var -5) var nil nil))
6
```

今度はうまくいきました，この時switchがどのように展開されたかというと，

```
(let ((g000 (plus var -5)))
     (cond ((geatrerp g000 0) var)
           ((zerop g000) nil)
           (t nil)))
```

のようになったのです．

最後に，gensym_f()はoblistに登録されないシンボルアトムを作るために，mk_sub()でシンボルアトムを作っていることに注意してください．このmk_sub()という関数は，oblistへの登録作業(intern)をおこないません．

● intern_f()　(134行〜151行)

oblistに登録されていないシンボルアトムをoblistに登録します．同じ印字名を持つシンボルアトムがすでにoblist中にある時はエラーになります．

● remob_f()　(153行〜166行)

oblistからシンボルアトムを取り除きます．もちろんシンボルアトムがなくなってしまうわけではありません．oblistから参照できなくなるだけです．この関数では引数のチェックをするだけで，実際の仕事は次の関数sub_remob()がおこなっています．

● sub_remob()　(168行〜201行)

oblistからなくなってしまっては困るシンボルアトムがいくつかあります．nil, lambdaなどは消えてもらっては困るのです．これらの不可欠なシンボルアトムでなければ，oblistを検索して，そのシンボルアトムのつながっているセルをoblistから削除します．

6.6.2 バッククォート

まず，リーダはバッククォート形式を次のように展開します．

```
^form    →  (backquote form)
,form    →  (comma form)
,@form   →  (atmark form)
```

したがって，

```
^(form1 ,form2 ,@form3)
```

のような文字列は，読み込まれた時点で

```
(backquote (form1 (comma form2) (atmark form3)))
```

という形式のS式に変換され，メモリに格納されます．さらにこのS式はevalに渡された時に評価を受け，通常の関数を用いたリストに変換されます．その処理をおこなうのが以下の4つの関数です．

● bq_f()　(203行〜216行)

バッククォート評価の最上位に位置する関数です．このbq_f()は，以下の規則でバッククォート形式を通常のS式に変換します．

(1) (backquote　<アトム>)は(quote　<アトム>)と等しい．
(2) (backquote　(comma　<S式>))は<S式>の中に存在するかもしれないバッククォート形式を評価するため(bqev　<S式>)を実行する．
(3) (backquote　(atmark　<S式>))は誤りである．
(4) (backquote　<リスト>)はリストの中に存在するかもしれないcomma，atmarkを処理するため，(bqapnd　<リスト>)を実行する．

● bqapnd()　(218行〜288行)

comma，atmarkを処理しながらリストを作ります．リストの最初をスタックに待避しなくてはならないため，先頭要素に対する処理と，後の部分に対する処理をおこなうループに分かれており，同じような処理が2つ並んでいます．

リストの作成は，元のバッククォート形式のリストを最初の要素から調べながら，別のセルにコピーを作っていきます．この時，

(comma　<S式>)

という要素に対しては，Fig. 6.10のように<S式>をbqev()で評価した結果を作成中のリストの要素としてコピーし，

(atmark　<S式>)

に対しては，Fig. 6.10のように<S式>をbqev2()で評価して得られるリスト(リストが得られなければエラー)を展開し，そのすべての要素を作成中のリストに組み込みます(Fig. 6.10)．atmarkはリストを展開するという約束上，ドット対のcdrにはできません．

● bqev()　(290行〜300行)

commaで指定されたS式を評価するための関数です．リストを評価する際に，リスト内のバッククォート形式を処理するためにbqapnd()を通してから評価する点が普通の評価と異なります．

● bqev2()　(302行〜320行)

atmarkで指定されたS式を評価し得られるリストのコピーを作ります．

また，comma，atmark は bqapnd()の中でラベルとして使われ処理されるためのもので，これらはバッククォート構文以外の場所で関数として使用されてはなりません．そこで，わざと次の2つの関数を用意し，バッククオート以外の所で用いられた場合のエラーチェックをおこなっています．

(backquote a (comma (car '(b c))) (atmark (list 'c 'd)))
bqev
bqapnd
(car '(b c))
eval
b
bqev2
bqapnd
(list 'c 'd)
eval
(c d)
複製
(c d)
bqapnd
(a b c d)

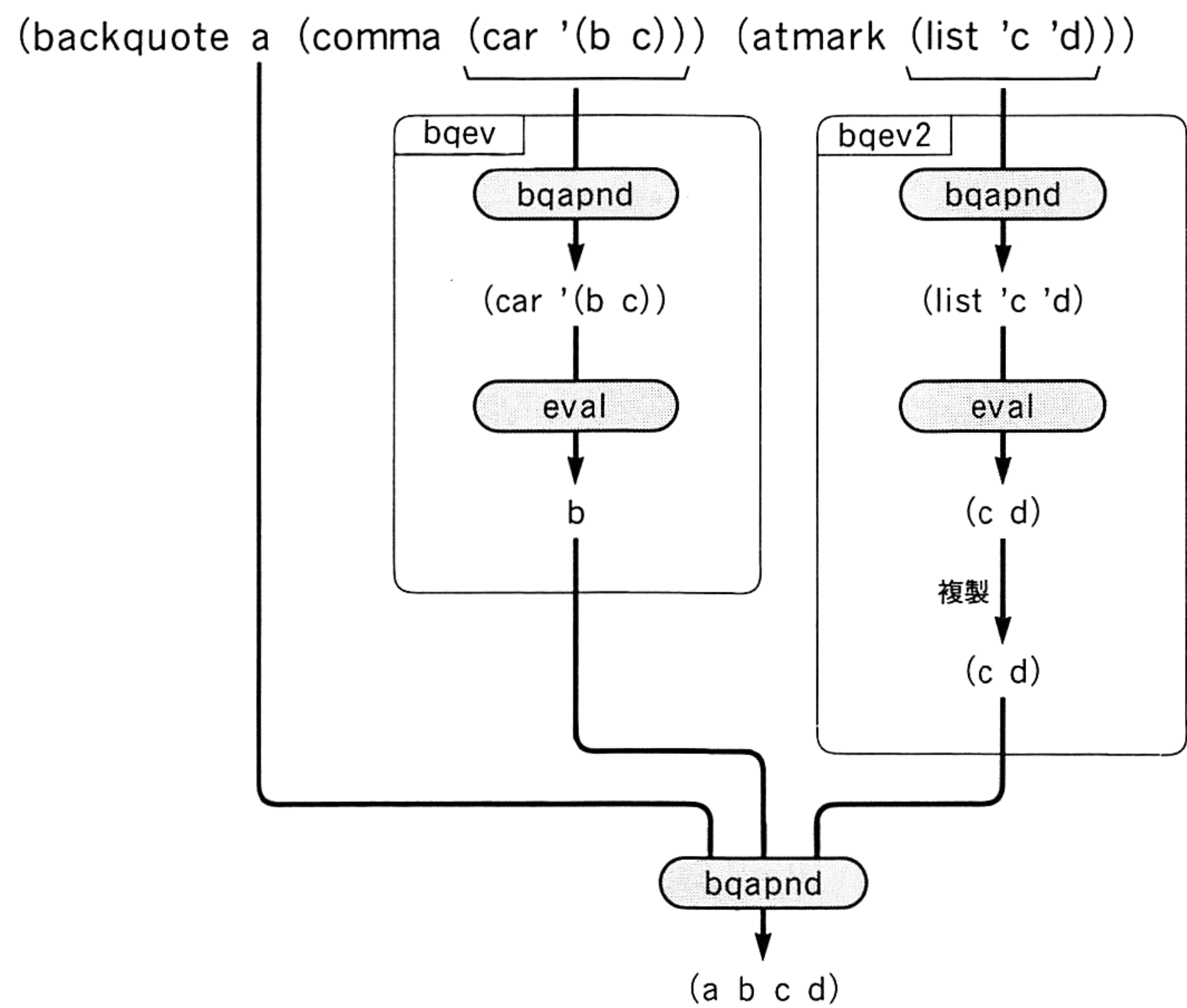



FIg. 6.10 バッククォートの評価

● comma_f() （322 行～326 行）

常にエラーを返します．

● atmark_f() （328 行～332 行）

常にエラーを返します．

6.6.3 マクロ機能などの追加に伴うプログラムの変更

以上の機能を追加するために，MADALTO バージョンのプログラムリストに，さらに次のような変更が必要になります．

● lisp.h （List 6.45）

マクロを表すコードと，関数がマクロ形式であることを検出するためのマスク，および5種類のエラーコードが追加されます．

● defvar.h/var.h (List 6.46/List 6.47)

バッククォートとマクロ関連のシステムアトムが4つ，テンプレート関連のシステムアトムが7つ追加されます．

● error.c (List 6.51)

追加されたエラーに対応するエラーメッセージがerr_msg［］の最後に追加されます．

● inisubr.c (List 6.53)

新しく追加された7つのLisp関数を定義するため，init6()が追加され，それに応じてini_subr()も変更されます．

6.6.4 シャープサインマクロ

〝'〟(クォート)がquoteの略記に用いられたのと同様に，〝#〟(シャープサイン)もリーダの機能の一部として有効な働きを持ちます．シャープサインは，その直後に続く文字によって，数種類の異なる動作をおこないます．Will o'Lispのシャープサインは，以下の5種類の機能を備えています．

#'form	……(function form)の略記
#:form	……シンボルアトムformはoblistに登録されない(internされない)
#o	……8進数入力
#x	……16進数入力
#¥	……文字コードによる1文字入力

これらのように，特定の展開作用(マクロ作用)を持つ文字をマクロ文字といいます．

これらのマクロを追加するのはread.cに若干の変更を加えれば済みます．以前より〝#〟はすでに特殊文字として登録されていますので，特殊文字を読み込んだ時の処理をする関数escopt()に〝#〟の処理をする関数option()を呼出す項を追加します．手順としては*quote*の追加をおこなった時と同じ要領です．

● option() (39行〜55行)

〝#〟の次にくる文字によって処理を場合分けします．

● read_o(), read_x() (57行～95行)

8進入力，16進入力をおこなう関数です．

● read_c() (105行～120行)

〝#¥nnn〞という文字列を読み込み，10進数nnnを文字コードとして持つような印字名のシンボルアトムを作成します．英数字，カタカナ，漢字以外のコントロールコードを印字名として持つシンボルアトムを作成するために用いられます．

6.6.5 read.c以外のマクロに伴う変更点

● dm_f() (List 6.54 3行～17行)

dmは，macro型の関数を定義するための関数です．この型の関数の定義は，

(macro <仮引数リスト> <本体1> <本体2> ……)

というmacro式をシンボルアトムのfptrに与え，そのアトムのftypeを_MACROにセットすることでおこなわれます．

● print.c (List 6.49)

oblistに登録されているシンボルアトムと登録されていないシンボルアトムを区別して表示するため，*print/prin1*出力時にoblistに登録されていないシンボルアトムの先頭に〝#：〞を付けてoblistに未登録であることを示します．ユーザーは〝#：〞によってoblistに登録されないシンボルアトムを自由に作ることができましたから，これでプリンタとリーダの間の同一性が保たれたことになります．

この機能は関数putstr()にoblistの検索と〝#：〞の表示の処理を付け加えるだけです．

● eval.c (List 6.50)

apply()の中にmacroの評価の部分が追加されます．このmacroの評価はlambda/nlambda式の評価とほとんど同じで，ただひとつ異なるのはbodyの最後の部分が2回の評価を受ける点です．1回目の評価でマクロの展開式が生成され，それをもう1度評価することにより，マクロの機能が実現されます．

6.6.6 MAXで追加されるその他の機能

●イニシャルプログラムローダ

これは，Lisp起動時に指定されたプログラムを自動的に読み込む機能です．main()内のinit()の呼出のあとに，ファイルinitial.lspを読み込む関数autoexe()を追加します．この関数はloadf()と同様の機能を持ちますが，対象となるファイルはinitial.lspただひとつです．このファイルがカレントディレクトリに見つかればそのファイル内のS式を読み込み評価します．

●トップレベルループの改造

トップレベルループにヒストリー機能を付けます．

+，++，+++の3つのシンボルアトムによって入力式のヒストリーを，*，**，***の3つのシンボルアトムによって評価後の出力式のヒストリーを記録しておきます．さらに，-によって現在の入力式を記録します．これは便利な機能で，前回の評価結果を使いたい時や，同じ処理を繰り返したい時など長いS式を繰り返し打込んだり，いちいち変数に代入する必要がありません．toplevel_f()が大分長くなりますが，増えるのは条件判断とヒストリーのシンボルアトムの値の移し換えだけです．

● 子プロセスの呼出

control.cのファイルに子プロセスを呼出す関数*exec*を追加します．これは，かなりコンパイラによって異なり，移植の際には書き換えが必要になります．プログラム中のものはMS-Cを用いていますので，他のコンパイラに移植されるかたは注意してください．

List 6.45 lisp.h 最終形(これまでの追加分もすべて含む)

```
 1: /*                                   */
 2: /*  Lisp Interpreter Header File     */
 3: /*                                   */
 4:
 5: #include    <stdio.h>
 6:
 7: /*                                    */
 8: /*  type and structure definition     */
 9: /*                                    */
10:
11: typedef unsigned char   uchar;
12: typedef uchar           *STR;
13:
14: typedef struct  cell {
15:     char    id;
16:     struct  cell    *car;
17:     struct  cell    *cdr;
18: } CELL;
```

```
19:
20: typedef CELL     *CELLP;
21:
22: typedef struct  atom {
23:     char    id;
24:     CELLP   value;
25:     CELLP   plist;
26:     STR     name;
27:     char    ftype;
28:     CELLP   fptr;
29: } ATOM;
30:
31: typedef ATOM     *ATOMP;
32:
33: typedef struct num {
34:     char    id;
35:     union body {
36:         struct num *ptr;
37:         long    fix;
38:         double  flt;
39:     } value;
40: } NUM;
41:
42: typedef NUM      *NUMP;
43:
44: typedef struct fstrct {
45:     char    mode;
46:     FILE    *ptr;
47: } FILE_S;
48:
49: /*                              */
50: /*  tag number and mask of      */
51: /*      id and function type    */
52: /*                              */
53:
54: #define _CELL    1
55: #define _ATOM    2
56: #define _FIX     3
57: #define _FLT     4
58:
59: #define _NFUNC   0x01     /* 0001 */
60: #define _SUBR    0x06     /* 0110 */
61: #define _EXPR    0x04     /* 0100 */
62: #define _FSUBR   0x02     /* 0010 */
63: #define _FEXPR   0x00     /* 0000 */
64: #define _MACRO   0x08     /* 1000 */
65:
66: #define _UD      0x01     /* 0001 */
67: #define _SR      0x02     /* 0010 */
68: #define _EA      0x04     /* 0100 */
69: #define _MC      0x08     /* 1000 */
70: #define _TR      0x10
71: #define NONMRK   (_UD | _SR)
72:
73: #define USED     0x80
74: #define FREE     (~USED)
75:
76: /*              */
77: /*  size of ... */
78: /*              */
79:
80: #define CELLSIZ      0x1800       /* 6550 (Max of large model) */
81: #define ATOMSIZ      0x0500       /* 1280 */
82: #define STRSIZ       0x1000       /* 4096 */
83: #define NUMSIZ       0x1000       /* 4096 */
84: #define STACKSIZ     0x0500       /* 1280 */
85: #define TABLESIZ     64
86: #define LINESIZ      100
```

```
 87: #define NAMLEN      100
 88: #define NFILES      10
 89:
 90: /*                  */
 91: /*  file open mode  */
 92: /*                  */
 93:
 94: #define READFILE    2
 95: #define WRITEFILE   1
 96: #define APPENDFILE  0
 97: #define IFREAD      2
 98: #define BINF        4
 99:
100: /*            */
101: /*  switches  */
102: /*            */
103:
104: #define ESCON   1
105: #define ESCOFF  0
106: #define ON      1
107: #define OFF     0
108:
109: #define TOP 0
110: #define UNDER   1
111:
112: #define TRUE    (-1)
113: #define FALSE   0
114:
115: /*                  */
116: /*  used for 'err'  */
117: /*                  */
118:
119: #define ERROK   (-1)
120: #define ERR     1
121: #define THROW   2
122: #define GO      3
123: #define RET     4
124:
125: /*               */
126: /*  error check  */
127: /*               */
128:
129: #define     ec  if(err)return(NULL)
130: #define     stackcheck  if(sp>=stacktop+STACKSIZ){error(STACKUP);return(NULL
     );}
131:
132: /*                */
133: /*  error number  */
134: /*                */
135:
136: #define NONERR  0   /* Non error */
137: #define STRUP   1   /* String area used up */
138: #define NUMUP   2   /* Number area used up */
139: #define ATOMUP  3   /* Atom area used up */
140: #define CELLUP  4   /* Cell area used up */
141: #define ULO     5   /* Unidentified Lisp Object */
142: #define PSEXP   6   /* Pseudo S-expression */
143: #define CTRLIN  7   /* Controll CHR. in the text */
144: #define UDF     8   /* Undefined function */
145: #define IFF     9   /* Illegal function form */
146: #define NEA     10  /* Not enough arguments */
147: #define IAA     11  /* Illegal argument--Atom required */
148: #define IAN     12  /* Illegal argument--Number required */
149: #define IAL     13  /* Illegal argument--List required */
150: #define IAAL    14  /* Illegal arg.--Atom or List required */
151: #define IAAN    15  /* Illegal arg.--Atom or Number required */
152: #define IALN    16  /* Illegal arg.--List or Number required */
153: #define IAF     17  /* Illegal argument--Fix Num required */
```

```
154: #define IAFL     18  /* Illegal argument--Float Num required */
155: #define ILS      19  /* Illegal structure (Illegal form) */
156: #define IASSL    20  /* Illegal A-list */
157: #define IPL      21  /* Illegal property list */
158: #define EIA      22  /* Environment is an Atom */
159: #define EHA      23  /* Environment has an Atom */
160: #define CCL      24  /* Condition Clause must be a List */
161: #define EOFERR   25  /* Unexpected EOF */
162: #define CCC      26  /* Cannot Change Constant value */
163: #define UNDEF    27  /* Error undefined */
164: #define STACKUP  28  /* Softwear stack used up */
165: #define NSG      29  /* No such go-label */
166: #define RWP      30  /* Return without Prog */
167: #define TTA      31  /* Throw Tag must be an Atom */
168: #define TWC      32  /* Throw without Catch */
169: #define ILV      33  /* Illegal Local Variable List */
170: #define FNA      34  /* File Not Available */
171: #define IFD      35  /* Illegal File Descripter */
172: #define FNO      36  /* File is not opend */
173: #define TMF      37  /* Too many Files */
174: #define FRO      38  /* File is Read Only */
175: #define FWO      39  /* File is Write Only */
176: #define CCF      40  /* Cannot Close File */
177: #define DIVZERO  41  /* Division by zero */
178: #define MOL      42  /* Mapped Object must be a List */
179: #define IBQ      43  /* Illegal Backquote Form */
180: #define CRS      44  /* Cannot remove System Atom */
181: #define AAE      45  /* Atom already exists */
182: #define ANE      46  /* Atom doesn't exist */
183: #define TMA      47  /* Too Many Arguements */
184:
185: /*                      */
186: /*  external variable   */
187: /*                      */
188:
189: #ifdef  MAIN
190: #include     "defvar.h"
191: #else
192: #include     "var.h"
193: #endif
```

List 6.46 defvar.h 最終形(これまでの追加分もすべて含む)

```
 1: /*                              */
 2: /*  define external variables   */
 3: /*                              */
 4:
 5: FILE_S   fp[NFILES];
 6: FILE     *cur_fpi, *cur_fpo;
 7:
 8: ATOMP    t, nil, lambda, eofread;
 9: ATOMP    prompt, stdprompt;
10: ATOMP    quote;
11: ATOMP    function, funarg, nlambda;
12: ATOMP    bq, comma, atmark, macro;
13: ATOMP    r1, r2, r3, e1, e2, e3, now;
14:
15: CELLP    oblist[ TABLESIZ ];
16:
17: CELLP    celltop, freecell;
18: ATOMP    atomtop, freeatom;
19: NUMP     numtop, freenum;
20: STR      strtop, newstr;
21:
22: uchar    oneline[ LINESIZ ];
23: STR      txtp;
24:
```

```
25: int      err, err_no;
26:
27: CELLP    *stacktop, *sp;
28: int      verbos;
29:
30: CELLP    throwlabel, throwval;
31:
32: int      trind = 1;
```

List 6.47 var.h 最終形(これまでの追加分もすべて含む)

```
 1: /*                                   */
 2: /*  list of external variables       */
 3: /*                                   */
 4:
 5: extern FILE_S   fp[NFILES];
 6: extern FILE     *cur_fpi, *cur_fpo;
 7:
 8: extern ATOMP    t, nil, lambda, eofread;
 9: extern ATOMP    prompt, stdprompt;
10: extern ATOMP    quote;
11: extern ATOMP    function, funarg, nlambda;
12: extern ATOMP    bq, comma, atmark, macro;
13: extern ATOMP    r1, r2, r3, e1, e2, e3, now;
14:
15: extern CELLP    oblist[ TABLESIZ ];
16:
17: extern CELLP    celltop, freecell;
18: extern ATOMP    atomtop, freeatom;
19: extern NUMP     numtop, freenum;
20: extern STR      strtop, newstr;
21:
22: extern uchar    oneline[ LINESIZ ];
23: extern STR      txtp;
24:
25: extern int      err, err_no;
26:
27: extern CELLP    *stacktop, *sp;
28: extern int      verbos;
29:
30: extern CELLP    throwlabel, throwval;
31:
32: extern int      trind;
```

List 6.48 read.c escopt(), ret_atom()を変更，マクロ用の関数を追加

```
 1: static CELLP escopt(level)
 2: int level;
 3: {
 4:     CELLP   mk_list(), error();
 5:     CELLP   formal(), option();
 6:     ATOMP   ret_atom();
 7:
 8:     switch(*txtp) {
 9:         case '(':
10:         case '[':   return mk_list(level);
11:         case '|':
12:         case '¥¥':  return (CELLP)ret_atom(ON);
13:         case '¥'':  return formal(level, quote);
14:         case '#':   return option(level);               ┐
15:         case '^':   return formal(level, bq);           │
16:         case ',':   if(*(txtp+1) == '@') {              │
17:                         ++txtp;                         ├ backquote形式の読み込み
18:                         return formal(level, atmark);   │
19:                     }                                   │
20:                     else return formal(level, comma);   ┘
```

```
21:         default:    return error(PSEXP);
22:     }
23: }
24:
25: static ATOMP ret_atom(mode)
26: int mode;
27: {
28:     char     nambuf[NAMLEN + 1];
29:     ATOMP    ap;
30:     ATOMP    old_atom(), mk_atom(), mk_sub();
31:
32:     getname(nambuf);
33:     if (!mode)  return mk_sub(nambuf); ………internされないシンボルの製作
34:     if ((ap = old_atom(nambuf)) == NULL)
35:         return mk_atom(nambuf);
36:     return ap;
37: }
38:
39: static CELLP option(level) ………#マクロに対する処理の割り振り
40: int level;
41: {
42:     CELLP    formal(), error();
43:     ATOMP    ret_atom(), read_c();
44:     NUMP     read_o(), read_x();
45:
46:     switch(*(++txtp)) {
47:         case '¥'':  return formal(level, function);
48:         case ':':   ++txtp;
49:                     return (CELLP)ret_atom(OFF);
50:         case 'o':   return (CELLP)read_o();
51:         case 'x':   return (CELLP)read_x();
52:         case '¥¥':  return (CELLP)read_c();
53:         default:    return error(PSEXP);
54:     }
55: }
56:
57: static NUMP read_o() ………8進入力
58: {
59:     double  l = 0;
60:     NUMP    np, newnum();
61:     CELLP   error();
62:
63:     if (*(++txtp) < '0' || *txtp > '7')
64:         return (NUMP)error(PSEXP);
65:     while (*txtp >= '0' && *txtp <= '7') {
66:         l = l*8 + *(txtp++) - '0';
67:     }
68:     if (l > 0x7FFFFFFF) l = -1;
69:     np = newnum();  ec;
70:     np->value.fix = l;
71:     return np;
72: }
73:
74: static NUMP read_x() ………16進入力
75: {
76:     int     c;
77:     double  l = 0;
78:     NUMP    np, newnum();
79:     CELLP   error();
80:
81:     ++txtp;
82:     if (!ishex(tolower(*txtp)))
83:         return (NUMP)error(PSEXP);
84:     while (ishex(c = tolower(*txtp))) {
85:         if (isdigit(c))
86:             l = l*16 + c - '0';
87:         else
88:             l = l*16 + c - 'a' + 10;
```

```
 89:         ++txtp;
 90:     }
 91:     if (l > 0x7FFFFFFF) l = -1;
 92:     np = newnum();  ec;
 93:     np->value.fix = (long)l;
 94:     return np;
 95: }
 96:
 97: static int ishex(c)
 98: char  c;
 99: {
100:     if (isdigit(c))              return TRUE;
101:     if (c >= 'a' && c <= 'f')   return TRUE;
102:     return FALSE;
103: }
104:
105: ATOMP read_c() ………"#¥○○○"の読み込み
106: {
107:     ATOMP   ap, old_atom(), mk_atom();
108:     char    nambuf[ LINESIZ ], *tp;
109:
110:     ++txtp;
111:     if(!(isdigit(*txtp)))
112:         return (ATOMP)error(PSEXP);
113:     for(tp = nambuf; isdigit(*txtp);)
114:         *tp++ = *txtp++;
115:     nambuf[ 0 ] = (char)atoi(nambuf);
116:     nambuf[ 1 ] = '¥0';
117:     if((ap = old_atom(nambuf)) == NULL)
118:         return mk_atom(nambuf);
119:     return ap;
120: }
121:
122: CELLP gensym_f()
123: {
124:     static int   i = 0;
125:     ATOMP   ap, mk_sub();
126:     char    nam[5];
127:
128:     sprintf(nam, "g%03d", i);
129:     ap = mk_sub(nam);    ec;
130:     if (++i > 999)  i = 0;
131:     return (CELLP)ap;
132: }
133:
134: CELLP intern_f(arg)
135: CELLP arg;
136: {
137:     CELLP   error();
138:     ATOMP   ap1, ap, old_atom();
139:
140:     if (arg->id != _CELL)   return error(NEA);
141:     ap1 = (ATOMP)arg->car;
142:     while (arg->id == _CELL) {
143:         if ((ap = ((ATOMP)(arg->car)))->id != _ATOM)
144:             return error(IAA);
145:         if (old_atom(ap->name) == NULL)
146:             intern(ap);
147:         else error(AAE);
148:         arg = arg->cdr;
149:     }
150:     return (CELLP)ap1;
151: }
152:
153: CELLP remob_f(arg)
154: CELLP arg;
155: {
156:     CELLP   error();
```

```
157:     ATOMP   ap;
158:
159:     if (arg->id != _CELL)   return error(NEA);
160:     ap = (ATOMP)arg->car;
161:     while (arg->id == _CELL) {
162:         sub_remob(arg); ec;
163:         arg = arg->cdr;
164:     }
165:     return (CELLP)ap;
166: }
167:
168: static sub_remob(arg) ……… oblist からの削除
169: CELLP arg;
170: {
171:     ATOMP   ap;
172:     CELLP   cp, error();
173:     unsigned int    i = 0;
174:
175:     if ((ap = ((ATOMP)(arg->car)))->id != _ATOM)
176:         return (int)error(IAA);
177:     if (ap == t || ap == nil || ap == quote || ap == lambda || ap == nlambda
    )
178:         return (int)error(CRS);
179:     if (ap == function || ap == funarg || ap == macro || ap == eofread)
180:         return (int)error(CRS);
181:     if (ap == r1 || ap == r2 || ap == r3 || ap == e1 || ap == e2)
182:         return (int)error(CRS);
183:     if (ap == e3 || ap == now || ap == prompt || ap == stdprompt)
184:         return (int)error(CRS);
185:     if (ap == bq || ap == comma || ap == atmark)
186:         return (int)error(CRS);
187:     i = hash(ap->name);
188:     if (oblist[i] == (CELLP)nil)    return (int)error(ANE);
189:     if ((cp = oblist[i])->car == (CELLP)ap) {
190:         oblist[i] = cp->cdr;
191:         return;
192:     }
193:     while (cp->cdr != (CELLP)nil) {
194:         if (cp->cdr->car == (CELLP)ap) {
195:             cp->cdr = cp->cdr->cdr;
196:             return;
197:         }
198:         cp = cp->cdr;
199:     }
200:     return (int)error(ANE);
201: }
202:
203: CELLP bq_f(args, env) ………バッククォート
204: CELLP args, env;
205: {
206:     CELLP   cp, bqev(), error(), bqapnd();
207:
208:     if(args->id != _CELL) return error(IBQ);
209:     if( (args = args->car)->id != _CELL )
210:         return args;
211:     else if( (cp = args->car) == (CELLP)comma )
212:         return bqev(args->cdr->car, env);
213:     else if( cp == (CELLP)atmark )
214:         return error(IBQ);
215:     else    return bqapnd(args, env);
216: }
217:
218: static CELLP bqapnd(args, env)
219: CELLP args, env;
220: {
221:     CELLP   newcell(), error(), bqev(), bqev2();
222:     CELLP   cp, cp2, *cpp;
223:
```

```
224:     stackcheck;
225:     cpp = ++sp;
226:     for(;;args = args->cdr) {
227:         if(args->car->id == _CELL) {
228:             if(args->car->car == (CELLP)atmark) {
229:                 *cpp = cp = bqev2(args->car->cdr->car, env); ec;
230:                 if (cp == (CELLP)nil) continue; ………, @nil はつながない
231:                 else while(cp->cdr->id == _CELL)
232:                         cp = cp->cdr;
233:             }
234:             else if (args->car->car == (CELLP)comma) {
235:                 *cpp = cp = newcell();  ec;
236:                 cp->car = bqev(args->car->cdr->car, env);
237:             }
238:             else {                                                  リストの先頭を
239:                 *cpp = cp = newcell();  ec;                         取り出す
240:                 cp->car = bqapnd(args->car, env);
241:             }
242:             ec;
243:         }
244:         else {
245:             *cpp = cp = newcell();  ec;
246:             cp->car = args->car;
247:         }
248:         break;
249:     }
250:     for(args = args->cdr;; args = args->cdr) {
251:         if(args->id != _CELL) {
252:             cp->cdr = args;
253:             return *(sp--);
254:         }
255:         else if(args->car == (CELLP)comma) {     最後のcar(ドット対)の処理
256:             cp->cdr = bqev(args->cdr, env);
257:             return *(sp--);
258:         }
259:         else if(args->car == (CELLP)atmark)
260:             return error(IBQ);
261:         if(args->car->id == _CELL) {
262:             if(args->car->car == (CELLP)atmark) {
263:                 cp->cdr = bqev2(args->car->cdr->car, env); ec;  , @formは評価し
264:                 while(cp->cdr->id == _CELL)                     展開する
265:                     cp = cp->cdr;
266:             }
267:             else if(args->car->car == (CELLP)comma) {
268:                 cp2 = newcell();    ec;
269:                 cp2->car = bqev(args->car->cdr->car, env);
270:                 cp->cdr = cp2;                                 , formは評価する
271:                 cp = cp->cdr;
272:             }
273:             else {
274:                 cp2 = newcell();    ec;
275:                 cp2->car = bqapnd(args->car, env);
276:                 cp->cdr = cp2;
277:                 cp = cp->cdr;
278:             }
279:             ec;
280:         }
281:         else {
282:             cp2 = newcell();    ec;
283:             cp2->car = args->car;      シンボルはそのまま
284:             cp->cdr = cp2;
285:             cp = cp->cdr;
286:         }
287:     }
288: }
289:
290: static CELLP bqev(args, env) ………, formのformの評価
291: CELLP args, env;
```

```
292: {
293:     CELLP   result, bqapnd(), eval();
294:
295:     if(args->id != _CELL)   return eval(args, env);
296:     else  {
297:         result = bqapnd(args, env); ec;
298:         return eval(result, env);
299:     }
300: }
301:
302: static CELLP bqev2(args, env) ………, @formでのformの評価
303: CELLP args, env;
304: {
305:     CELLP   cp, newcell(), bqev(), error();
306:
307:     args = bqev(args, env); ec;
308:     if(args == (CELLP)nil)  return (CELLP)nil;
309:     if(args->id != _CELL)   return error(IAL);
310:     stackcheck;
311:     *(++sp) = cp = newcell();   ec;
312:     forever {                                   ┐
313:         cp->car = args->car;                    │
314:         if(args->cdr->id != _CELL) break;       │
315:         cp->cdr = newcell();    ec;             ├ 展開用にcopyをとる
316:         cp = cp->cdr;                           │
317:         args = args->cdr;                       │
318:     }                                           ┘
319:     return *(sp--);
320: }
321:
322: CELLP comma_f(args, env)         ┐
323: CELLP args, env;                 │
324: {                                │
325:     return error(IBQ);           │
326: }                                │
327:                                  ├ baqkquote形式以外では‶,″, ‶, @″は使えない
328: CELLP atmark_f(args, env)        │
329: CELLP args, env;                 │
330: {                                │
331:     return error(IBQ);           │
332: }                                ┘
```

List 6.49 print.c pri_atom(), putstr()を変更

```
 1: pri_atom(cp, mode)
 2: CELLP cp;
 3: int mode;
 4: {
 5:     switch (cp->id) {
 6:         case _FIX:  fprintf(cur_fpo, "%ld", ((NUMP)cp)->value.fix);
 7:                     break;
 8:         case _FLT:  fprintf(cur_fpo, "%.6g", ((NUMP)cp)->value.flt);
 9:                     break;
10:         case _ATOM: putstr(mode, (ATOMP)cp); ………印字名でなく, シンボルそのものを渡す
11:                     break;
12:         default:    error(ULO);
13:     }
14: }
15:
16: static putstr(mode, ap)
17: int mode;
18: ATOMP   ap;
19: {
20:     ATOMP   ap2, old_atom();
21:     STR     tp = ap->name;
22:
23:     if (mode == ESCOFF)
```

```
24:             fprintf(cur_fpo, "%s", tp);
25:         else if (*tp == '¥0')
26:             fprintf(cur_fpo, "||");
27:         else {
28:             if ((ap2 = old_atom(tp)) == NULL || ap2 != ap)
29:                 fprintf(cur_fpo, "#:");
30:             if (num(tp))
31:                 fputc('¥¥', cur_fpo);
32:             do {
33:                 if (iskanji(*tp) && iskanji2(*(tp+1))) {
34:                     fputc(*tp++, cur_fpo);
35:                     fputc(*tp++, cur_fpo);
36:                 }
37:                 else if (!isprkana(*tp)) {
38:                     fprintf(cur_fpo, "#¥¥%03d", *tp++);
39:                 }
40:                 else {
41:                     if (isesc(*tp))
42:                         fputc('¥¥', cur_fpo);
43:                     fputc(*tp++, cur_fpo);
44:                 }
45:             } while (*tp != '¥0');
46:         }
47: }
```

オブリストにないシンボルには，"#:"をつける
後ろのap2!=apは，オブリストにあるシンボルと同じ印字名を持ちながら，それ自身はオブリストにないシンボルのためのもの

List 6.50 eval.c apply()を変更

```
 1: CELLP apply(func, args, env)
 2: CELLP func, args, env;
 3: {
 4:     CELLP   (*funcp)(), bodies, result = (CELLP)nil;
 5:     CELLP   bind(), error();
 6:     char    funtype = ~_TR;
 7:
 8:     switch (func->id) {
 9:         case _ATOM:
10:             funtype = ((ATOMP)func)->ftype;
11:             if (funtype & _UD)
12:                 return error(UDF);
13:             if (funtype & _TR)
14:                 trarg((ATOMP)(*++sp = func), args);
15:             if (funtype & _SR) {
16:                 funcp = (CELLP (*)())((ATOMP)func)->fptr;
17:                 if (funtype & _EA)
18:                     result = (*funcp)(args);
19:                 else
20:                     result = (*funcp)(args, env);
21:                 break;
22:             }
23:             func = ((ATOMP)func)->fptr;
24:         case _CELL:
25:             if (func->cdr->id != _CELL)
26:                 return error(IFF);
27:             if (func->car == (CELLP)funarg)
28:                 result = apply(func->cdr->car, args, func->cdr->cdr->car);
29:             else if (func->car == (CELLP)lambda
30:                   || func->car == (CELLP)nlambda) {
31:                 bodies = func->cdr->cdr;
32:                 stackcheck;
33:                 *++sp = bind(func->cdr->car, args, env); ec;
34:                 for (; bodies->id == _CELL; bodies = bodies->cdr) {
35:                     result = eval(bodies->car, *sp); ec;
36:                 }
37:                 --sp;
38:             }
39:             else if (func->car == (CELLP)macro) {
40:                 bodies = func->cdr->cdr;
```

```
41:                 stackcheck;
42:                 *++sp = bind(func->cdr->car, args, env); ec;
43:                 for(;bodies->id == _CELL;bodies = bodies->cdr) {   } マクロの評価
44:                     result = eval(bodies->car, *sp); ec;
45:                 }
46:                 result = eval(result, env); ………最後の評価結果を
47:                 --sp;                            もう一度評価する
48:             }
49:             break;
50:         default:
51:             return error(IFF);
52:     }
53:     ec;
54:     if (funtype & _TR)
55:         trval((ATOMP)*sp--, result);
56:     return result;
57: }
```

List 6.51 error.c err_msg[]の最終形

```
 1: static STR errmsg[] = {
 2:     "I don't know what happend",
 3:     "String area used up",
 4:     "Number area used up",
 5:     "Atom area used up",
 6:     "Cell area used up",
 7:     "Unidentified Lisp Object",
 8:     "Pseudo S expression",
 9:     "Ctrl character sneaks in",
10:     "Undefined function",
11:     "Illegal function form",
12:     "Not enough arguments",
13:     "Illegal argument--Atom required",
14:     "Illegal argument--Number required",
15:     "Illegal argument--List required",
16:     "Illegal arg.--Atom or List required",
17:     "Illegal arg.--Atom or Number required",
18:     "Illegal arg.--List or Number required",
19:     "Illegal argument--Fix Num required",
20:     "Illegal argument--Float Num required",
21:     "Illegal structure",
22:     "Illegal association list",
23:     "Illegal property list",
24:     "Environment is an atom",
25:     "Environment has an atom",
26:     "Condition Clause must be a List",
27:     "Unexpected EOF",
28:     "Can't Change Constant value",
29:     "Unidentified error",
30:     "Software stack used up",
31:     "No such go-label",
32:     "Return without Prog",
33:     "Throw Tag must be an Atom",
34:     "Throw without Catch",
35:     "Illegal Local Variable List",
36:     "File Not Available",
37:     "Illegal File Descripter",
38:     "File is not opened",
39:     "Too many Files",
40:     "File is Read Only",
41:     "File is Write Only",
42:     "Can't Close File",
43:     "Division by Zero",
44:     "Mapped Object must be a List",
45:     "Illegal Backquote Form",
46:     "Can't remove System Atom",
```

```
47:     "Atom already exists",
48:     "Atom doesn't exist",
49:     "Too Many Arguements",
50: };
```

List 6.52 main.c main(), toplevel_f(), mk_sys_atoms(), greeting()を変更, autoexe()を追加

```
 1: main()
 2: {
 3:     greeting();
 4:     init();
 5:     r1->value = r2->value = r3->value = (CELLP)nil; } テンプレートの初期化
 6:     e1->value = e2->value = e3->value = (CELLP)nil; }
 7:     autoexe();
 8:     forever {
 9:         if (err == ERR) pri_err((CELLP)nil);
10:         reset_err();
11:         toplevel_f();
12:         reset_stdin();
13:     }
14: }
15:
16: toplevel_f()
17: {
18:     CELLP   *arg;
19:     CELLP   read_s(), eval(), error();
20:
21:     stackcheck;
22:     arg = ++sp;
23:
24:     forever {
25:         now->value = read_s(TOP);
26:         if (now->value == (CELLP)eofread)   break;
27:         ec;
28:         if (now->value == (CELLP)r1)    now->value = (CELLP)r1->value; }
29:         if (now->value == (CELLP)r2)    now->value = (CELLP)r2->value; }
30:         if (now->value == (CELLP)r3)    now->value = (CELLP)r3->value; } テンプ
31:         if (now->value == (CELLP)e1)    now->value = (CELLP)e1->value; } レート
32:         if (now->value == (CELLP)e2)    now->value = (CELLP)e2->value; } 機能
33:         if (now->value == (CELLP)e3)    now->value = (CELLP)e3->value; }
34:         *arg = eval(now->value, (*arg = (CELLP)nil));
35:         switch (err) {
36:             case ERROK:
37:             case ERR:   return((err = ERROK));
38:             case THROW: return (int)error(TWC);
39:             case GO:    return (int)error(NSG);
40:             case RET:   return (int)error(RWP);
41:         }
42:         print_s(*arg, ESCON);
43:         ec;
44:         fputc('¥n', cur_fpo);
45:         if (isatty(fileno(cur_fpi)))
46:             fputc('¥n', cur_fpo);
47:         r3->value = r2->value;  }
48:         r2->value = r1->value;  }
49:         r1->value = now->value; }
50:         e3->value = e2->value;  } テンプレート用シンボルの内容を変更する
51:         e2->value = e1->value;  }
52:         e1->value = *arg;       }
53:     }
54:     sp--;
55: }
56:
57: static mk_sys_atoms()
58: {
59:     ATOMP   mk_atom();
```

```
 60:     CELLP   quote_f(), funct_f();
 61:     CELLP   bq_f(), comma_f(), atmark_f();
 62:
 63:     mk_nil();
 64:     t = mk_atom("t");
 65:     lambda = mk_atom("lambda");
 66:     eofread = mk_atom("EOF");
 67:     prompt = stdprompt = mk_atom("% ");
 68:     quote = mk_atom("quote");
 69:     quote->ftype = _FSUBR;
 70:     quote->fptr = (CELLP)quote_f;
 71:     nlambda = mk_atom("nlambda");
 72:     function = mk_atom("function");
 73:     function->ftype = _FSUBR;
 74:     function->fptr = (CELLP)funct_f;
 75:     funarg = mk_atom("funarg");
 76:     bq = mk_atom("backquote");
 77:     bq->ftype = _FSUBR;
 78:     bq->fptr = (CELLP)bq_f;
 79:     comma = mk_atom("comma");
 80:     comma->ftype = _FSUBR;
 81:     comma->fptr = (CELLP)comma_f;
 82:     atmark = mk_atom("atmark");
 83:     atmark->ftype = _FSUBR;
 84:     atmark->fptr = (CELLP)atmark_f;
 85:     macro = mk_atom("macro");
 86:     r1 = mk_atom("+");
 87:     r2 = mk_atom("++");
 88:     r3 = mk_atom("+++");
 89:     e1 = mk_atom("*");
 90:     e2 = mk_atom("**");
 91:     e3 = mk_atom("***");
 92:     now = mk_atom("-");
 93: }
 94:
 95: static greeting()
 96: {
 97:     fprintf(stdout,"¥n");
 98:     fprintf(stdout,"¥tSuperceding Lisp Interpreter¥n");
 99:     fprintf(stdout,"¥t              M A X¥n");
100:     fprintf(stdout,"¥t  Will o'Lisp Version 1.00¥n");
101:     fprintf(stdout,"¥t        (C)  1986, Mar¥n¥n");
102:     fprintf(stdout,"¥t    Created by PIN & Zdo¥n¥n");
103: }
104:
105: static autoexe() ………イニシャルプログラムローダー
106: {
107:     FILE    *lfp, *fopen();
108:
109:     if( (lfp = fopen("initial.lsp", "r")) == NULL )
110:         return FALSE;
111:     reset_err();
112:     cur_fpi = fp[3].ptr = lfp;
113:     fp[3].mode = READFILE;
114:     toplevel_f();
115:     fclose(lfp);
116:     fp[3].ptr = NULL;
117:     fputc('¥n', cur_fpo);
118: }
```

List 6.53 inisubr.c ini_subr()を変更，init6()を追加

```
 1: ini_subr()
 2: {
 3:     init0();
 4:     init1();
 5:     init2();
 6:     init3();
 7:     init4();
 8:     init5();
 9:     init6();
10: }
11:
12: static init6()
13: {
14:     CELLP dm_f(), getd_f(), putd_f();
15:     CELLP intern_f(), remob_f(), gensym_f();
16:     CELLP exec_f();
17:
18:     defsubr("dm",       dm_f,       _FSUBR);
19:     defsubr("getd",     getd_f,     _SUBR);
20:     defsubr("putd",     putd_f,     _SUBR);
21:     defsubr("intern",   intern_f,   _SUBR);
22:     defsubr("remob",    remob_f,    _SUBR);
23:     defsubr("gensym",   gensym_f,   _SUBR);
24:     defsubr("exec",     exec_f,     _SUBR);
25: }
```

List 6.54 control.c dm_f(), getd_f(), putd_f(), exec_f()を追加

```
 1: #include    <process.h>
 2:
 3: CELLP dm_f(args, env) ………macro式の定義をする
 4: CELLP args, env;
 5: {
 6:     CELLP form, cons(), error();
 7:     ATOMP mac;
 8:
 9:     if (args->id != _CELL || args->cdr->id != _CELL)
10:         return error(NEA);
11:     if ((mac = (ATOMP)args->car)->id != _ATOM)
12:         return error(IAA);
13:     form = cons((CELLP)macro, args->cdr); ec;
14:     mac->ftype = _MACRO;
15:     mac->fptr = form;
16:     return (CELLP)mac;
17: }
18:
19: CELLP getd_f(args)
20: CELLP args;
21: {
22:     CELLP cp, getd(), error();
23:
24:     if (args->id != _CELL)
25:         return error(NEA);
26:     if (args->car->id != _ATOM)
27:         return error(IAA);
28:     cp = getd((ATOMP)args->car); ec;
29:     return cp;
30: }
31:
32: static CELLP getd(func)
33: ATOMP func;
34: {
35:     NUMP    newnum();
36:     CELLP   cp, newcell();
37:
```

```
 38:     if (func->ftype & _UD)
 39:             return (CELLP)nil;
 40:     if (func->ftype & _SR) { ………(F)SUBRのときはftypeとfptrのコンスを返す
 41:             cp = newcell(); ec;
 42:             cp->car = (CELLP)newnum(); ec;
 43:             ((NUMP)(cp->car))->value.fix = (long)func->ftype;
 44:             cp->cdr = (CELLP)newnum(); ec;
 45:             ((NUMP)(cp->cdr))->value.fix = (long)func->fptr;
 46:             return (CELLP)cp;
 47:     }
 48:     return func->fptr;
 49: }
 50:
 51: CELLP putd_f(args)
 52: CELLP args;
 53: {
 54:     CELLP def, label, error(), eval();
 55:     ATOMP func;
 56:
 57:     if (args->id != _CELL || args->cdr->id != _CELL)
 58:         return error(NEA);
 59:     if ((func = (ATOMP)args->car)->id != _ATOM)
 60:         return error(IAA);
 61:     if ((def = args->cdr->car)->id != _CELL)
 62:         return error(IAL);
 63:     if (def->car->id == _FIX) {
 64:         if (def->cdr->id != _FIX) return error(IAF);
 65:         func->ftype = (char)((NUMP)(def->car))->value.fix;
 66:         func->fptr = (CELLP)((NUMP)(def->cdr))->value.fix;
 67:         return def;
 68:     }
 69:     if (def->car == (CELLP)funarg) {
 70:         if (def->cdr->car->id != _CELL)
 71:             return error(IFF);
 72:         label = def->cdr->car->car;
 73:     }
 74:     else
 75:         label = def->car;
 76:     if (label == (CELLP)lambda)
 77:             func->ftype = _EXPR;
 78:     else if (label == (CELLP)nlambda)
 79:             func->ftype = _FEXPR;
 80:     else if (label == (CELLP)macro)
 81:             func->ftype = _MACRO;
 82:     else
 83:             return error(IFF);
 84:     return func->fptr = def;
 85: }
 86:
 87: CELLP exec_f(args) ………子プロセスの実行
 88: CELLP args;
 89: {
 90:     CELLP   error();
 91:     int     forkerr;
 92:     int     i = 1;
 93:     ATOMP   process;
 94:     char    mybuf[127], *mybufp, *myargv[10], *strcpy();
 95:
 96:     if (args->id != _CELL)
 97:         return(error(NEA));
 98:     if ((process = (ATOMP)args->car)->id != _ATOM)
 99:         return(error(IAA));
100:     myargv[0] = process->name;
101:     args = args->cdr;
102:     mybufp = mybuf;
103:     while (args->id == _CELL) {                                         ┐
104:         if (i >= 10) return error(TMA);                                 │
105:         if (args->car->id != _ATOM) return(error(IAA));                 │ mybufにコマン
106:         myargv[i++] = strcpy(mybufp, ((ATOMP)args->car)->name);         │ ド列をつくる
```

```
107:         while (*mybufp++ != '¥0');
108:         args = args->cdr;
109:     }
110:     myargv[i] = NULL;
111:     forkerr = spawnvp(P_WAIT, process->name, myargv);
112:
113:     fprintf(cur_fpo, "¥nerror : %d¥n", forkerr);
114:     return (CELLP)t;
115: }
```

APPENDIX

1. 各バージョンにおけるプログラムリストの変更点のまとめ
2. Lattice C への移植について
3. MSX-CおよびLSI Cへの移植について
4. コンパイル手順

1. 各バージョンにおけるプログラムリストの変更点のまとめ

6章では，段階を追った機能拡張によって，最終的なLisp処理系を完成させるという手順をとりました．その際，本来ならば各ステップごとにすべてのソースリストを掲載したかったのですが，ページ数の関係上それは不可能であるため，それぞれ〝前バージョンとの差分〟のみを掲載してあります．結果として最終的なMAXバージョンのプログラムリストが各節に分散してしまい，プログラム全体の見通しが少々悪くなってしまいました．

そこで，Will o'Lispを構成するファイルと本書に掲載したリストの番号の対応を，Table. A. 1にまとめておくことにします．この中で，それぞれのリストに付けられた記号には，次のような意味を持っています．

無印 ： 前のバージョンに対する追加あるいは変更のためのリスト
△ ： 前のバージョンのリストは必要としない独立したリストだが，後に追加あるいは変更を受けるリスト
◎ ： 前のバージョンのリストは必要とせず，後にも追加変更されない独立したリスト

	UNDEAD	MADI	MALOR	CALFO	MONTINO	MADALTO	MAX
lisp.h	5.1	6.1	6.12	6.19	6.27	6.36	◎ 6.45
defvar.h	5.2	6.2	6.13	6.20		6.37	◎ 6.46
var.h	5.3	6.3	6.14	6.21		6.38	◎ 6.47
read.c	5.4	6.4		6.22			6.48
print.c	5.5						6.49
gbc.c	5.6	◎ 6.5					
eval.c	5.7	6.6				△ 6.39	6.50
error.c	5.8	6.7	6.15	6.23	6.28	6.40	6.51
main.c	5.9	6.8	6.16	6.24	6.29	6.41	6.52
inisubr.c	5.10	△ 6.9	6.17	6.25	6.30	6.42	6.53
fun.c	5.11	6.10			6.31	6.43	
control.c	5.12	6.11			6.32		6.54
calc.c	5.13				6.33		
iofunc.c	5.14			◎ 6.26			
prog.c			◎ 6.18				
pred.c					◎ 6.34		
str.c					◎ 6.35		
mapper.c						◎ 6.44	

Table.A.1　掲載リスト一覧

また，最終的なMAXバージョンのWill o'Lispを構成するために必要なリストには，アミをかけてあります．そして，lisp.h，defvar.h，var.hの3つのヘッダ・ファイルについては，〝大は小を兼ねる〟のことわざどおり，初めからMAXバージョン用の最終形リストを用いても問題はありません．

2. Lattice Cへの移植について

本書に掲載したプログラムは，Microsoft C ver.3.00.17を用いてコンパイルおよび実行をおこないましたが，プログラムの中でMicrosoft Cに特有のライブラリ関数あるいはLattice Cの旧バージョン(2.1xバージョン)に備えられていなかったライブラリ関数をいくつか使用しています．そのため，Lattice Cを用いて本書のプログラムをコンパイルする場合には，次のようなプログラムの変更が必要となります．

●spawnvp()

この関数はexec_f()の中で，子プロセスを呼び出すために使用されるもので，Microsoft Cに特有のものです．Lattice Cを使用する場合には，同等の機能を持つforkvp()を使用してください．

●isatty()

この関数は〝isatty(fileno(cur_fpi))〟という形で，現在の入力先がコンソールであるか否を判断するために用いられています．Lattice2.1Xバージョンにはisatty()関数がありませんので，この部分を〝現在の入力先がstdinであるか否か〟という判断に変えて代用します．具体的には，

```
if (isatty(fileno(cur_fpi))) ……
```

と記述された部分を，すべて，

```
if (cur_fpi == stdin) ……
```

という形に変更してください．ただしこの変更によって，Will 'o Lispの起動時に，

```
A>lisp < example.lsp
```

というようにリダイレクトをかけて，stdinをファイルに向け，その内容を読み込ませることはできなくなります．

また，これに関連して，main.cファイル中のreset_stdin()関数の定義，およびそれを呼び出している部分は，すべて削除してください．

●signal()

Lattice Cでは，ver.3.00になってsignal()が採用され，Ctrl-Cによる割り込みをCプログラムの中で捕まえられるようになりました．たとえば割り込み処理をおこなう関数の中で"Breakエラー"を発生してリターンするようにしておけば，Lispが無限ループに入ってしまった場合に，それをCtrl-Cで中断させLispのトップレベルに戻ることも可能です．ただし，本書のプログラムでは，Ctrl-C割り込みの処理は，直ちに終了メッセージを表示してDOSへ戻るのみとしました．

Lattice C ver.2.1Xを使用する場合には，main.cファイルに含まれているsignal()の呼び出し，およびprocess.hをインクルードしている部分は削除してください．

●unsigned char型

Lattice C ver.2.1Xには，unsigned char型が存在しません．ただし，char型は常にunsignedとして扱われるため，uchar型の定義をすべてchar型に変更すれば問題はありません．最も簡単な方法は，lisp.hの中の

```
#typedef unsigned char uchar;
```

と定義している部分を，

```
#typedef char uchar;
```

と書き換えてしまうことでしょう．

●fprintf()による浮動小数点表記

print.cファイルの中で，実数型の数値の表示のために，fprintf()の"%#.6g"という表示フォーマットを用いましたが，この"#"を用いた指定はLattice C ver.2.1Xの仕様にはありません（Lattice C ver.3.00になって採用された）．Lattice C ver.2.1Xを用いる場合には，この部分を"% g"と書き換えれば，同じ結果が得られます．

3．MSX-CおよびLSI Cへの移値について

8ビットマシン上のCコンパイラの例として，LSI CおよびMSX-Cへの移植を取り上げます．これらの処理系は，構文的な処理能力は標準のCとほとんど変わりませんが，やはり8ビットマシンであることの制限は大きく，かなりの機能を削除しなくてはWill o'Lispは動作しません．具体的には，以下のような点を変更する必要があります．

● メモリ容量の制限

メモリ容量が50〜64Kバイトしかないため，その中でMAXバージョンを動作させることは不可能です．本書のプログラムでいえば，CALFOバージョンまでが限界でしょう．ただしその場合でも，セル領域の大きさ(CELLSIZ)を1000，アトム領域の大きさ(ATOMSIZおよびNUMSIZ)を500程度に減らし，またiofunc.cの中で非常に大きな場所を占めているformat関数の定義(List 6.25の50行，List 6.26の346行以降）を省く必要があります．

● 非再帰関数宣言の利用

MSX-CやLSI Cでは，積極的にnonrec宣言を取り入れ，直接あるいは間接に再帰呼び出しされない関数には非再帰関数の宣言をおこなうと，メモリ効率がたいへんに向上します．ただしこの時，自分の中でeval()を呼び出しているfsubr型の関数は，潜在的に再帰性を備えていることに注意してください．たとえば，

```
(setq a (setq b 'c))
```

を評価すると，内側のsetqは外側のsetqの中で評価されることになります．これに対し，subr型の関数は引数が評価されてから渡されるため，nonrec宣言をおこなってかまいません．

● 数値の取り扱い

MSX-CやLSI Cには，float型とlong型が存在しません．したがって数値アトムはint型のみとなります．float型に関する記述はすべて削除し，long型はint型に変更してください．

例：

```
 1: CELLP add1_f(arg)…………………MS-C版のadd1_f()，これを下記のリストのように変更する
 2: CELLP arg;
 3: {
 4:     char    c;
 5:     NUMP    np, newnum();
 6:     CELLP   error();
 7: 
 8:     if (arg->id != _CELL)
 9:         return error(NEA);
10:     if ((c = arg->car->id) != _FIX && c != _FLT)
11:         return error(IAN);
12:     np = newnum();  ec;
13:     if (c == _FIX)
14:         np->value.fix = ((NUMP)(arg->car))->value.fix + 1;
15:     else {
16:         np->id = _FLT;
17:         np->value.flt = ((NUMP)(arg->car))->value.flt + 1;
18:     }
19:     return (CELLP)np;
20: }
```

```
 1: CELLP add1_f(arg)…………………LSI C, MSX-C版のadd1_f()
 2: CELLP arg;
 3: {
 4:     char     c;
 5:     NUMP     np, newnum();
 6:     CELLP    error();
 7:
 8:     if (arg->id != _CELL)
 9:         return error(NEA);
10:     if (arg->car->id != _FIX)
11:         return error(IAN);
12:     np = newnum();  ec;
13:     np->value.fix = ((NUMP)(arg->car))->value.fix + 1;
14:     return (CELLP)np;
15: }
```

●シンボル名の長さ制限

グローバルなシンボルは6文字までしか識別されません．たとえば，〝close()〟と〝close_f()〟という2つの関数名は，どちらも〝close@〟というラベルに変換されてしまいます．これを避けるために，次のような宣言をlisp.hの中に含めておく必要があります．

```
 1: #define fgetc   getc
 2: #define fputc   putc
 3: #define getch   getchar
 4:
 5: #define quote   qt
 6: #define oblist  ob
 7: #define cur_fpi c_fpi
 8: #define cur_fpo c_fpo
 9: #define dirin   din
10: #define dirout  dout
11: #define vervos  vb
12: #define prompt  pt
13: #define close_f cls_f
14: #define rem_mark_num rmn
```

4．コンパイル手順

各種のモジュールをコンパイルしてWill o'Lispを生成する場合，メモリモデルとスタックサイズの指定に注意しなければなりません．セルやアトムが使用するデータ領域が大きいため，メモリモデルにはラージモデルを指定する必要があります．また，スタックサイズはラージモデルで取り得る，ほぼ最大の60000バイトを指定してください．たとえば，Microsoft Cでコンパイルする場合には，次のような書式になります．

書式　cl ／AL ／Fe〈ファイル名1〉〈ファイル名2〉 ／link ／stack：60000

／AL ………………………ラージモデルの指定スイッチ
／Fe ………………………実行型ファイル名の指定スイッチ
／stack： …………………スタックサイズの指定スイッチ
〈ファイル名1〉……………実行型ファイル名
〈ファイル名2〉……………コンパイルの対象ファイル名

コンパイルの対象となるファイルの指定は，どのバージョンのWill o'Lispを生成するかによって異なりますが，たとえば，UNDEADバージョンのWill o'Lispをコンパイルするのであれば，List 5.1からList 5.14のプログラムを用意して，〈ファイル名2〉には，〝*.c〟を指定します．他のバージョンのコンパイル作業もコンパイルの際に指定するファイルの違いを除けば，同様におこなうことができます．詳しくは，Table A.1を参照してください．

Fig. A.1に，UNDEADバージョンのWill o'Lispのコンパイル手順を示します．

```
A>set↵
COMSPEC=A:¥COMMAND.COM
PATH=A:¥BIN………コンパイラの処理系の入っているパスの設定
INCLUDE=a:¥include………インクルードファイルのパス
LIB=a:¥lib………ライブラリのパス

A>b:↵

B>cd ¥src↵………ドライブBの"¥SRC"にUNDEADバージョンのソース・プログラムが置かれている

B>dir /w↵………ディレクトリの内容を確認する

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥src

.                  ..                 LISP      H      DEFVAR    H      VAR       H
CALC      C        CONTROL   C        ERROR     C      EVAL      C      FUN       C
GBC       C        INISUBR   C        IOFUNC    C      MAIN      C      PRINT     C
READ      C
         16 個のファイルがあります
  1164288 バイトが使用可能です
```

```
B>cl /AL /Feundead *.c /link /stack:60000 ⏎………コンパイラの起動
Microsoft (R) C Compiler  Version 3.00.17
Copyright (C) Microsoft Corp 1984. 1985.  All rights reserved.

CALC.C
CONTROL.C
ERROR.C
EVAL.C
FUN.C
GBC.C
INISUBR.C
IOFUNC.C
MAIN.C
PRINT.C
READ.C

Microsoft 8086 Object Linker
Version 3.01 (C) Copyright Microsoft Corp 1983, 1984. 1985

Object Modules [.OBJ]: CALC.OBJ CONTROL.OBJ ERROR.OBJ EVAL.OBJ FUN.OBJ GBC.OBJ I
NISUBR.OBJ IOFUNC.OBJ MAIN.OBJ PRINT.OBJ READ.OBJ
Run File [CALC.EXE]: UNDEAD.EXE/NOI
List File [NUL.MAP]: NUL
Libraries [.LIB]: /STACK:60000 ;

B>dir *.exe ⏎………実行型ファイルの確認

 ドライブ B: のディスクにはボリュームラベルがありません
 ディレクトリは B:¥src

UNDEAD    EXE     41836  86-08-19  14:24
        1 個のファイルがあります
  1091584 バイトが使用可能です

B>undead ⏎………UNDEADを実行する

        Superceding Lisp Interpreter  ┐
                 U N D E A D          │
          Will o'Lisp Version 0.00    │ オープニングメッセージ
               (C)  1986. Feb         │
                                      │
            Created by PIN & Zdo      ┘

%(quit)⏎………終了


        May the force be with you!
                        --From Will o'Lisp with Love.
B>
```

Fig.A.1 Will o'Lispのコンパイル手順

参 考 文 献

・プログラミング言語C
　（B. W. Kernighan, D. M. Ritchie共著，石田晴久訳，共立出版）
・C言語入門
　（L. Hancock, M. Krieger共著，アスキー出版局監訳，アスキー出版局）
・Cプログラムブック I
　（打越浩幸，濱野尚人，梅原系共著，アスキー出版局）
・Cプログラムブック II
　（打越浩幸，濱野尚人，梅原系共著，アスキー出版局）
・情報処理シリーズ4　Lisp
　（P. H. Winston, B. K. P. Horn共著，白井良明，安部憲広共訳，培風館）
・COMMON LISP
　（Guy L. Steele Jr. 他著，後藤英一監訳，井田昌之訳，共立出版）
・コンピュータサイエンス大学講座1　Lisp入門　――システムとプログラミング
　（中西正和著，近代科学社）
・電子計算機のプログラミング＝7　LISP入門
　（黒川利明著，培風館）
・Lisp入門
　（池田一夫，馬場史郎共著，啓学出版）
・Common Lisp入門　1～3
　（湯浅太一，萩谷昌己，bit 1985年4～6月号，共立出版）
・Common Lispのデータ型とその周辺　1～5
　（井田昌之，bit 1985年8～12月号，共立出版）

Cプログラムブック III

1986年9月11日　初版発行
定価2,800円

著　者　小西弘一・清水 剛
発行者　塚本慶一郎
発行所　株式会社アスキー
　〒107 東京都港区南青山6-11-1スリーエフ南青山ビル
振　替　東京4-161144
TEL (03)486-7111（大代表）
情報TEL (03)498-0299（ダイヤルイン）
出版営業部 TEL (03)486-1977（ダイヤルイン）

表紙担当　郷　啓子
CTS　株式会社電算プロセス
印刷　図書印刷株式会社

ISBN4-87148-200-6 C3055 ¥2800E

Cプログラムブック I

打越浩幸
濱野尚人
梅原　系　共著

定価2,800円(〒300円)

C言語が得意とするシステムプログラミングを中心とした，本格的プログラム集です．C言語の特徴を活かしたプログラミングの手法を詳細に解説することで，C言語の修得を容易にした一冊．C言語による実践的なプログラム開発をめざすユーザーに最適です．

内容は，UNIXライクなMS-DOS用フィルタコマンド群の整備，簡単なテキストエディタの作成例，高級アセンブラと呼ばれるC言語の特徴を活かした機械語ツールの実現，アセンブリプログラムとのリンクによる汎用グラフィックパッケージと，それを使った3次元ワイヤフレームプログラム，さらに複雑なアルゴリズムを構造的に記述できることを活かしてForthライクなインタープリタの制作など，たった20行の（有用な）プログラムから，中規模プログラムのモジュール別開発例までを，さまざまなC言語開発のコツをまじえながら，やさしく実用的に解説しています．

目次：基本ツール／エディタ／機械語ツール／グラフィックス／インタープリタ／APPENDIX（他の処理系への移植について）

CプログラムブックI

アスキー・ディスクアルバム[3]

Cプログラムブック I

DISK ALBUM 3

MS-DOS版
5インチ2DD／定価4,800円
8インチ2D／定価5,300円
(〒400円)

＊CプログラムブックIに掲載された
すべてのソフトウェアをディスクに収録

Cプログラムブック II

打越浩幸
濱野尚人
梅原　系　共著

定価2,800円(〒300円)

C言語について，すでにCプログラムブックIなどで文法をはじめとするプログラミングの基本的な知識を学び終わった方のために，さらに一歩進んだプログラムを制作するための手法や開発例などを解説しました．C言語による本格的なプログラム開発の実際を，ソースリストの詳細な解説から，分割コンパイルによるモジュール別開発／設計の考え方までを通して紹介しています．

最初に，C言語と非常に密接な関係にあるUNIX上のコマンドMAKEのMS-DOS版を解説することで，大規模プログラムの開発に必須である分割コンパイルを紹介しています．さらに，このMS-DOS用MAKEを使って，より本格的なグラフィックス（レイ・トレーシング）のプログラムを作成します．そして最後に，MS-DOS用のコンパイラの作成例を紹介し，C言語を含むコンパイラ一般に対する，より深い理解を可能にしています．C言語の可能性に挑戦するための一冊．

目次：MAKE－分割コンパイル支援ツール－／SHADO－レイ・トレーシング－／REDA－コンパイラ－／APPENDIX（コンパイル環境と移植について）

CプログラムブックII

アスキー・ディスクアルバム[11]

Cプログラムブック II

DISK ALBUM 11

MS-DOS版
5インチ2DD／定価5,300円
5インチ2HD／定価5,800円
8インチ2D／定価5,800円
(〒400円)

＊CプログラムブックIIに掲載された
すべてのソフトウェアをディスクに収録

ISBN4-87148-200-6 C3055 ¥2800E 定価 2,800円